FreeBSD™ JOURNAL

Sept/Oct 2017

FreeBSD *VERSUS* Linux

Microservices to Monoliths

FreeBSD *VS.* **Linux: ZFS**

FreeBSD is not a **Linux Distro**

A Comparison of **Unix Sandboxing Techniques**

The Evolution of **FreeBSD Governance**

FreeBSDJournal.org

Table of Contents

FreeBSD™ vs. Linux

COLUMNS & DEPARTMENTS

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Copy Editor • Annaliese Jakimides

Art Director • Dianne M. Kischitz
dianne@freebsdjournal.com

Office Administrator • Michael Davis
davism@freebsdjournal.com

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
5757 Central Ave., Suite 201, Boulder, CO 80301
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Board

"Isn't FreeBSD just a Linux distro?"

is a frequent question from those who come to FreeBSD from the Linux world. Readers of this publication know the response is "No," but they also know that's not the entire answer. FreeBSD isn't just a different code base; it's also a different philosophy and community, and in this issue, we take the question head on and show, through a set of articles, that FreeBSD is not a Linux distro. The title of the first piece, written by your humble Editor in Chief, first occurred to me when I was asked to give a talk about FreeBSD to a group of engineers at Digital Ocean in early 2015. Many of us who work on FreeBSD shy away from any mention of that other open-source operating system, but I felt it was time to take on the topic, and did so in a talk that was recorded at the event (https://www.youtube.com/watch?v=wwbO4eTieQY). After that first presentation, I found there were requests for the talk to be given at other venues and with more updates. Then FreeBSD Foundation Director Philip Paeps gave an updated version of the talk in his own style at Rootconf this year (https://www.youtube.com/watch?v=ps67ECyh0sM). When we began to put together this issue on FreeBSD vs. Linux, it was obvious that the talk needed to be turned into an article.

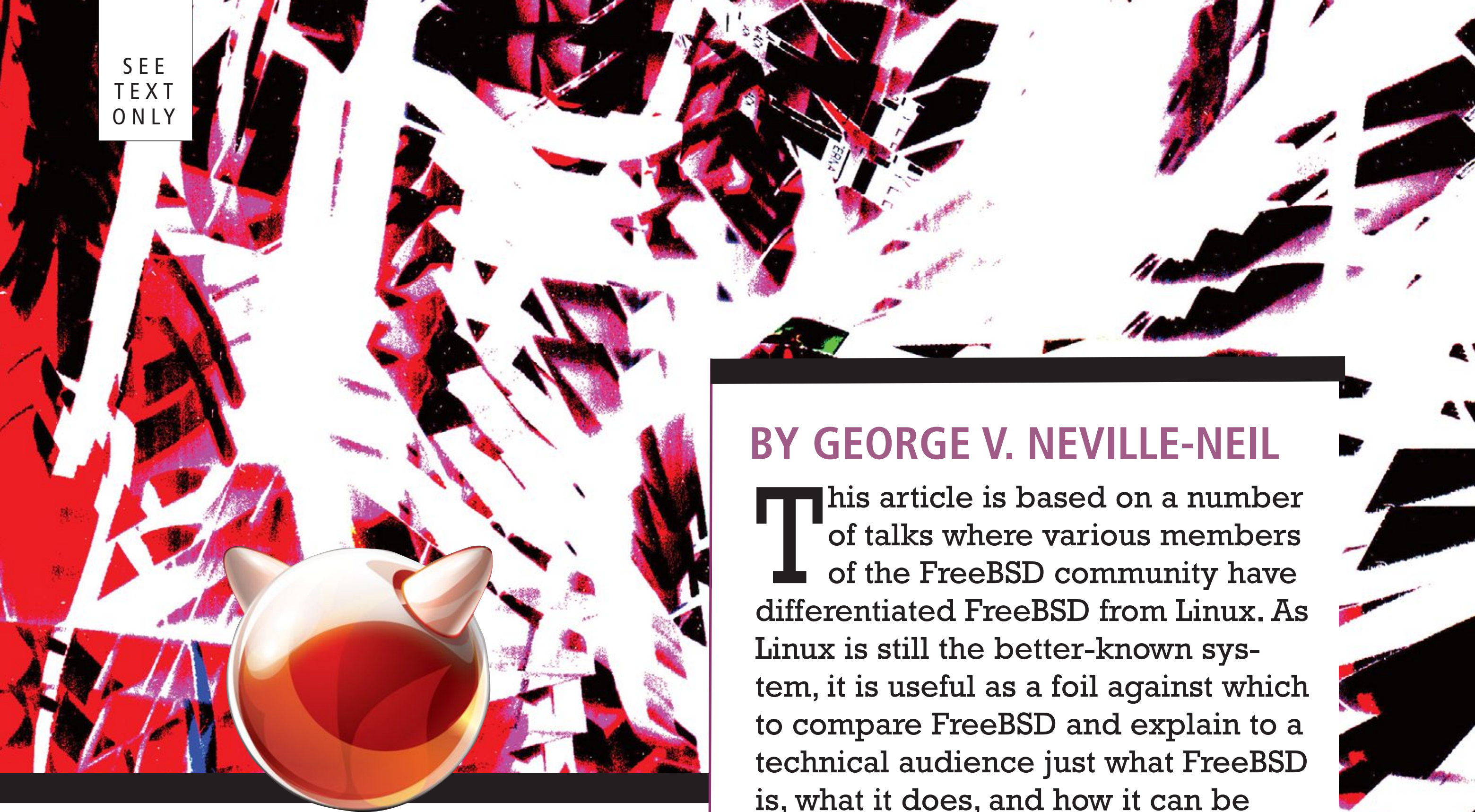
While an overview of how the two systems are different is interesting, we wanted to dig more deeply into how FreeBSD was not Linux. FreeBSD vs. Linux: ZFS by Allan Jude covers the only open-source, well-supported, filesystem for petabyte storage. With the demise of BRTFS on Linux, the only viable alternative is OpenZFS, and OpenZFS runs perfectly on FreeBSD. Jonathan Anderson takes us through the various techniques and technologies used for sandboxing on Unix systems, with special attention paid to Capsicum, the native capability system on FreeBSD. And Kirk McKusick, who was instrumental in helping define the early governing structure of the project, and Benno Rice, one of the current members of the core team, have coauthored the definitive article on how the FreeBSD Project is run.

Sit at a conference lunch or dinner with a bunch of FreeBSD developers and mention the lack of Control-T, aka SIGINFO, on Linux, and you will get a very loud chorus about how they all are completely annoyed by the lack of this feature on Linux. It seems like such a small difference, but to working programmers, it really matters. Benedict Reuschling has contributed a fine, short article on this feature of large importance.

And lastly, our themed articles are rounded out with an experience piece on Microservices by Dave Cottlehuber. Microservices are all the rage and they can easily be implemented on FreeBSD. Dave tells us how.

We trust that when you've finished this issue, you will have a definitive reply for your coworkers should they ask, "Isn't FreeBSD just a Linux distro?"

George Neville-Neil
President of the *FreeBSD Foundation* Board of Directors



SEE TEXT ONLY

FreeBSD is Not a LINUX DISTRO

BY GEORGE V. NEVILLE-NEIL

This article is based on a number of talks where various members of the FreeBSD community have differentiated FreeBSD from Linux. As Linux is still the better-known system, it is useful as a foil against which to compare FreeBSD and explain to a technical audience just what FreeBSD is, what it does, and how it can be applied to build a variety of systems that require an operating system. Since this article appears in the *FreeBSD Journal*, I expect readers are familiar with FreeBSD, and so I have organized the article as a set of talking points that can be referenced when you're trying to explain FreeBSD to colleagues or company management.

FreeBSD Is a Complete System

FreeBSD is a complete, open-source operating system that includes all the sources and tools to enable the consumer of the operating system to rebuild the entire system from the supplied sources. The completeness of the system is one of the key differentiators between FreeBSD and Linux. Linux is actually just the operating system kernel—a large piece of code—but not a complete system. To build a Linux system, a number of tools, packages, and libraries need to be installed alongside the sources. With a FreeBSD system, users can rebuild their entire systems the moment after FreeBSD is installed onto their computers. The \emph{self hosting} aspect of the FreeBSD system should not be overlooked because it is a core part of the philosophy of FreeBSD, the empowerment of the user. A complete FreeBSD system comes not only with the source code, but also with extensive documentation on all aspects of the system, including the built-in commands, library, and kernel APIs. The goal is to make users of FreeBSD as productive as possible once the system has been installed.

The operating system is only a small component of most modern systems. The majority of the software involved in building and deploying any significant system is not a component of the operating system, but instead is part of the user space software that runs on top of the operating system. FreeBSD supports third-party software via the ports and packages system. Most users will interact only with the packages system, which contains pre-packaged versions of software such as web servers and browsers, high-performance math libraries, language compilers and interpreters, and nearly any other open-source tool that is actively being maintained. Packages are built from the ports system, which is a large, hierarchically organized collection of Makefiles that knows from whence to retrieve software and how to build that software so that it runs on FreeBSD. The current set of ports encompasses more than 24,000 software packages. Consumers of FreeBSD can add their own software packages to the ports system as a way of building their own customized, installable version of FreeBSD.

Who Is the End User?

A Linux distro is a way of packaging an operating system kernel along with a set of applications into a system that can be installed by an end user. For many years Linux distros have been targeted for either desktop or server installations. A common example is the Ubuntu distro which comes in two configurations, one for servers and the other for desktops.

From FreeBSD's point of view, the end user is most often an engineer or software developer who will use FreeBSD to create something new, based around the FreeBSD system. Consumers of FreeBSD come in all shapes and sizes, and over the years, other open-source systems have been developed around FreeBSD, which are then consumed for particular purposes, such as pfSense for firewalls, FreeNAS for storage appliances, and TrueOS (formerly PC-BSD) for desktops and laptops. It is these systems that appear closer to a "distro" than FreeBSD itself.

Alongside these open-source systems, many companies have taken all, or part, of FreeBSD and used it as the basis of their own products. Apple, NetApp, Isilon, Juniper, and Sony, to name only a few, have used FreeBSD as the basis of successful products in desktop, mobile, storage, networking, and gaming. Each of these engineering consumers has been able to treat FreeBSD as a collection of libraries that can be combined to create an operating system to support their own products, and in this case, the end user isn't a consumer per se, but rather an engineer who is building a product for an end user.

Why Do People Use FreeBSD?

What motivates someone to use FreeBSD? There are five key reasons that people decide to use FreeBSD, and these are: our history of innovation, great tools, mature release model, documentation, and business-friendly license, all of which differentiate FreeBSD from a Linux distro.

FreeBSD is a child of the original Berkeley Software Distribution, the Unix variant that was developed at the University of California at Berkeley during the 1970s and 1980s. Virtual Memory, the Fast File System, Sockets API, and TCP/IP are all innovations that occurred in the original BSD system. FreeBSD has continued that tradition of innovation, producing high-performance software that was an early adopter of multicore systems, 10G, 40G, and 100G Ethernet, and the LLVM compiler suite.

LLVM is an example of how FreeBSD works with and integrates great tools. The LLVM compiler suite has led to an order of magnitude increase in interesting compiler technologies, technologies that just were not possible with the GNU compilers. LLVM has been particularly strong in the use of new compiler techniques to enhance security, leading to both research and deployment of more secure systems. With LLVM as our default compiler, all of these innovations have appeared in FreeBSD before appearing in other operating systems.

Anyone who has tried to produce a series of products on top of a Linux distro has run up against the inconsistencies in the Linux release model. Between one minor release and another, the Linux kernel APIs are not stable; in fact, Linux is committed to not having stable kernel APIs. That may be fine for those who want to only run the tip of the tree, but it is a disaster if you are a company trying to field products and upgrade them over time.

The FreeBSD release model requires that long-term branches, those numbered 9.x, 10.x, 11.x, maintain kernel API stability throughout their lifetimes. This commitment to stability means that products built using FreeBSD 10 can continue to be upgraded until the end of life of the 10 branch. New features and bug fixes will appear throughout the lifetime of the branch, but they will only appear if they do not change the way in which already established APIs work. These concepts are often wrapped up in the idea of the principle of least astonishment (POLA), whereby we try to not surprise our consumers, and where we clearly delineate the circumstances under which breaking changes are allowed to occur, such as across major release branches.

FreeBSD's documentation is second to none, and is, in fact, often referenced by those who are looking for generic information about other Unix systems and how they work. The manual pages not only cover the traditional areas of the system, such as commands (Section 1), system calls (Section 2), and libraries (Section 3), but also the kernel APIs themselves are documented in Section 9 of the manual pages. *The FreeBSD Handbook* acts as an overarching reference document to the system as a whole and covers higher-level topics relating to systems administration and software development.

Open-source systems are not just software; they also encompass a philosophy, most often described in the license they choose to use. The two-clause BSD license used by FreeBSD embodies the FreeBSD philosophy, which might be summed up as: use our code and don't sue us. The license is short, a mere 200 words, and is easy to understand. The GPLv2 used in Linux is over 2,900 words, and requires a lawyer to understand all of the ramifications of working with it.

Community Organization

Each open-source project has its own unique form of organization. Linux adheres to the "big high stomper" model, whereby Linus and his various lieutenants control access to the source tree. Linus is the leader because he started the project, and his lieutenants maintain their position in the project because they were chosen by Linus.

The FreeBSD Project is organized by and for those who commit to the source tree, and has no single owner or dictator. A committer is simply a person with commit access to the central source code server. Three types of commit bits exist, one each for documentation, port, and the source tree. There is no wall between these three types of access; they exist simply to differentiate the areas in which people might be doing most of their work. A source bit is granted to those who are working on the built source of the system either in the kernel or its associated tools. Documentation bits are held by those who are working on manual pages as well as projects such as the *FreeBSD Handbook*, the living document that describes FreeBSD as a whole and which goes far beyond what is contained in the manual pages. The ports system is maintained by ports committers, who have ports bits. Any one person may have any or all of these bits, and, indeed, those who have worked on FreeBSD for a number of years often wind up with all three of these commit bits.

The process of being granted a commit bit is relatively straightforward:

1 Alice interacts with the project, often by submitting patches to the system sources, ports, or documentation.

2 Until Alice has her own commit bit, her changes must be committed by a current committer, Bob.

3 After some time, Bob realizes that Alice could be committing her changes on her own, and he proposes Alice for a commit bit.

4 Bob emails the appropriate team to propose Alice for a commit bit. If Alice is working on the source, then Bob will contact the core team, but if she is working on documentation, Bob will contact the docs team, or, if she is working on one or more ports, he'd contact the ports team.

5 The core, documentation, and ports teams can then vote on the proposed commit bit, and if the bit is granted, Bob would become Alice's mentor.

6 For some period of time Bob will have to approve all of Alice's commits to the tree, a process that requires Alice to check her changes with Bob, via the code review system, https://reviews.freebsd.org.

7 Once Bob is happy that Alice can get by on her own, he releases her from mentorship. Alice can now go on to mentor new committers on her own.

These seven steps are how the FreeBSD Project continues to bring new blood into the project.

The core team is one of the components of the mentorship process. The core team, consisting of nine members, is elected every two years to help run the project, but the core team does not dictate what the committers work on. Core exists to facilitate the project and occasionally steps in to mediate disagreements that may arise between committers. Core also sponsors other teams, such as the ports, documentation, security, and release engineering by granting hats. Within the project, a particular responsibility is referred to as a hat, and the head of the team is thought of as wearing that hat. A more complete description of the governance of the project is given in this issue by McKusick and Rice.

Modern Features

While philosophy, community organization, and licenses are important differentiators between FreeBSD and Linux, there are also excellent technological reasons for working with FreeBSD. FreeBSD has many modern features that are not present in any Linux distro, including the UFS and ZFS filesystems, DTrace, and Capsicum.

FreeBSD has always had a modern implementation of the Fast File System, originally designed and built for BSD, and continuously upgraded over the last 30 years to keep pace with changing storage technology. The latest version, UFS2, includes support snapshots, as well as journaled soft updates. Journaled soft updates make recovering from a system failure—one where the integrity of the filesystem must be verified—quick and painless. Prior to the inclusion of soft updates, the process of checking a filesystem, carried out by the fsck program, might take hours on a large disk, or even longer with a multi-terabyte disk. Journaled soft updates remove the need to check over the entire disk, meaning that even an 8T drive can be made ready within seconds after a system reboot.

Large storage deployments, those that contain tens of disks and petabytes of data, are addressed using the Zetabyte Filesystem (ZFS). First designed and implemented as part of Solaris, ZFS was imported into FreeBSD in the early 2000s and quickly gained popularity in systems that required a fully functional volume manager backed by the latest RAID techniques. ZFS in FreeBSD remains fully up-to-date with the code maintained in the OpenZFS project, and several of the ZFS developers now work directly on the code in FreeBSD

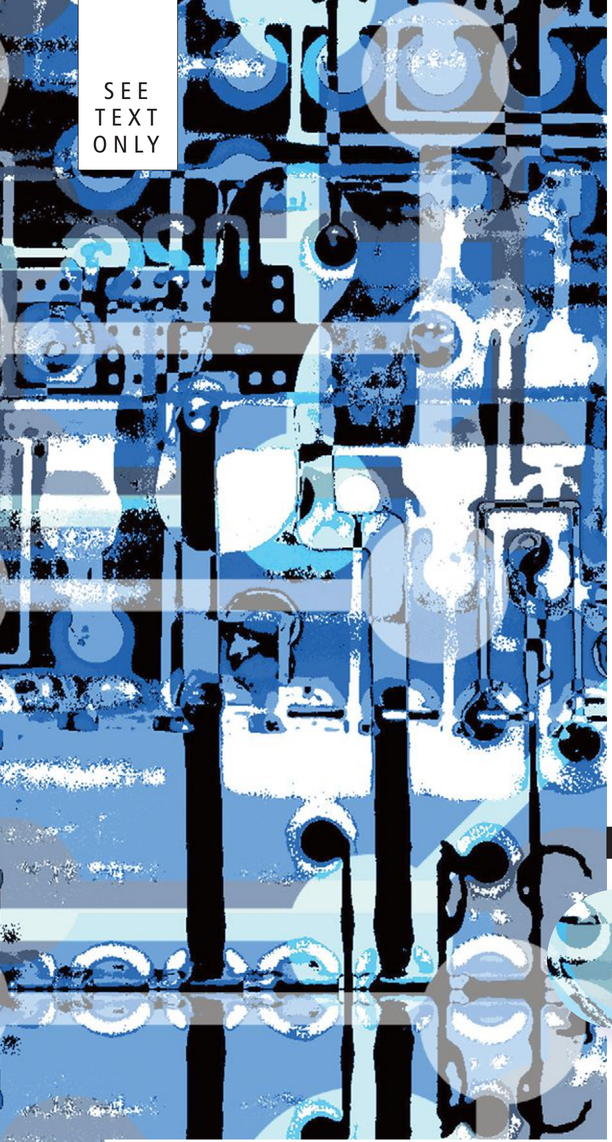
Another excellent technology to come out of Sun's Solaris group is DTrace, which was ported to FreeBSD in 2008, and which is now being maintained and updated actively within the FreeBSD source tree. DTrace gives programmers and systems administrators complete system transparency, allowing users to see inside any function call within the kernel or within a program running on FreeBSD, without the need for the original source code.

FreeBSD has always had a pragmatic approach to security, bringing in features that had broad applicability in securing the system. One recent addition, Capsicum, is a good example of this pragmatism. Capsicum is a modern capability system that was built with and for FreeBSD. Capabilities come out of research that was done in the 1970s, but there has never been a practical, widely fielded, capability system in a general-purpose operating system until Capsicum was integrated into FreeBSD. Capsicum helps developers build more-secure software by providing a lightweight framework for the compartmentalization of software. Robert Watson, who built Capsicum, points out, "Without compartmentalization, one vulnerability means that the entire application—and all the data it has access to—are available to the attacker. With compartmentalization, software is still vulnerable, but attackers must work harder to find and exploit many more vulnerabilities before they gain the full rights of the user."

Conclusion

The goal of this article was to give you a way of explaining to others how FreeBSD is not a Linux distro. We presented differentiators in five key areas, including: technical innovation, tooling, release model, documentation, and the business-friendly BSD license. Whether you're reading this as a consumer of FreeBSD or as someone who uses FreeBSD to produce some other, technical artifact, we're sure that you also have ways in which you see FreeBSD as being distinctly different from a Linux distro. We encourage readers to write the *Journal* to share their ideas. If we have a good response, we'll publish a brief, follow-up article so that your ideas can help others in the FreeBSD community. ●

GEORGE V. NEVILLE-NEIL works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, networking and time protocols. He is the coauthor with Marshall Kirk McKusick and Robert N. M. Watson of *The Design and Implementation of the FreeBSD Operating System*. For over 10 years he has been the columnist better known as Kode Vicious. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler and currently lives in New York City.



FreeBSD VS. Linux: ZFS

BY ALLAN JUDE

OpenZFS is available on many platforms, including FreeBSD and Solaris derivatives like IllumOS, as well as Mac OS X and Linux. However, not all of the functionality is available on the latter platforms. The FreeBSD Project has fully adopted ZFS, putting significant effort into integrating it with the system and management tools to make ZFS a seamless part of the OS, rather than a bolted-on extra. OpenZFS is better integrated, instrumented, and documented on FreeBSD than on any of the various Linux distros.

Why Use ZFS?

It is not that other filesystems are bad; they just make the mistake of trusting your storage hardware to return your data when you ask for it. As it turns out, hard drives are pretty good at that, but pretty good is often not good enough. ZFS is the only open-source, production-quality filesystem that can not only detect but correct the errors when a disk returns incorrect data. By combining the roles of filesystem and volume manager, ZFS is also able to ensure your data is safe, even in the absence of one or more disks, depending on your configuration. ZFS doesn't trust your hardware; it verifies that the correct data was returned from each read from the disk.

The primary design consideration for ZFS is the safety of the data. Every block that is written to the filesystem is accompanied by a checksum of the data, stored with the other metadata. That metadata block also has a checksum, as does its parent, all the way up to the top-level block, called the uber block. When the ZFS filesystem is mounted, it examines the available array of uber blocks and selects the newest one with a valid checksum. When combined with the copy-on-write feature, this means that in the event of a power failure or system crash, ZFS will still have a consistent view of the filesystem; any operations that were in progress, but did not complete, are rolled back to keep the filesystem in pristine shape. This means no need for a long filesystem check after an unexpected shutdown.

Every time a block of a file is read from a ZFS filesystem, the data returned by the disk is checksummed, and that checksum is verified against the one stored in the metadata. If the results differ, this means the disk has returned incorrect data. ZFS will detect this, keeping a count of such errors for each disk, as this may be a sign of impending disk failure. If your ZFS filesystem is configured with redundancy, like mirrors or RAID-Z, this parity information will be used to reconstruct the incorrect block, and write the repaired data back to the disk. If there is no redundancy and the data cannot be recovered, an error will be returned. This allows the operating system to stop an application from using invalid data, which may cause it to crash or do the wrong thing.

ZFS provides a great many other features, including transparent compression, advanced tiered caching, snapshots, deduplication, replication, delegation, boot environments, and, soon, even "at-rest" encryption. ZFS was designed to make the storage administrator's life easier by making adding capacity as easy as possible. ZFS also simplified the management of quotas and reservations, the management of network file-sharing, the delegation of permissions, and all of the other common tasks that make up a storage admin's day. All of this is managed from an intuitive command line interface, designed to be used by both human hands and automated with scripts.

Boot Environments

While the biggest feature of OpenZFS is the data-integrity guarantees it provides, the next most useful is the concept of boot environments. This feature allows a device to have multiple concurrent OS installations and easily flip between them at boot time. With the copy-on-write nature of ZFS, these additional images often consume very little storage space. All that is required is to have the root filesystem located on ZFS, and some integration with the boot loader. These features have been available in FreeBSD for years, and are constantly being refined and improved.

On a mobile computing platform, this flexibility and stability guarantee can be exceptionally useful. Imagine just a few minutes before your big presentation, you discover last week's update has caused some problem with the presentation software. Reboot, select last week's boot environment; now the OS and applications have been rolled back to the known working state, but your data files, home directory, etc., are still the most current version. Another reboot and you are back to the most up-to-date system image. The entire system is now effectively under a rudimentary type of version control. Fork your system and try an experiment, safe in the knowledge you can always flip back to the working system with a quick reboot.

Your separate system images do not need to be copy-on-write clones of each other. They can be entirely separate stand-alone images. Switch between the stable release and a developmental snapshot with ease, all while sharing your user data files. Need to test your latest work on every supported release of FreeBSD? No sweat. With some extra work, it is even possible to multiboot other operating systems that support OpenZFS.

At ScaleEngine we use this mechanism to distribute customized golden images of FreeBSD. After building and tweaking FreeBSD just as we like it, a replication stream is generated with the zfs send command, and redirected to a file. Production servers then fetch that image over HTTPS, and feed it into zfs recv, creating a new boot environment. Then zfsbootcfg configures that new boot environment to be used for the next boot (only). If there is something wrong with the image, such that it does not boot properly, or crashes, a power cycle will see the system come back up on the original system image. A script running in the new system image can make that image the new default if the system remains up for 10 minutes and the network connectivity requirements are met. This procedure allows us to safely upgrade remote systems all over the world with little chance of things requiring additional human intervention. If the new image is not satisfactory, a single command changes the default to one of the previous images and then a reboot puts the system back into operation. This all makes it extremely easy to test developmental images on a small fraction of our production fleet without any special handling.

Another feature that is coming soon to OpenZFS will see repeated system failures result in the system booting into a special debugging/rescue image. In environments like Amazon's EC2, where there is no console support, if the system does not come up fully, due to a boot failure or repeated panic, there is no easy way to repair the system, short of mounting its drive in a different EC2 instance. With this new feature, a counter will be incremented at each boot, and only if the system remains up long enough, will a script in the image reset this counter to zero. If the counter exceeds the threshold, the new boot will see it load the rescue image instead and summon an operator to debug and resolve the issue.

Disk Encryption

FreeBSD has a high-performance, full disk encryption system called GELI. It has long been used in combination with ZFS to create fully encrypted pools. This required special handling of the boot process, since the loader needed to load the kernel from unencrypted storage. In early 2016, I integrated GELI support into the bootstrap and loader to allow booting from a zpool with no part of the filesystem unencrypted.

The closed-source version of ZFS, now from Oracle, has supported encryption for some time, but this feature has never been available in open-source ZFS.

SSD TRIM

FreeBSD is currently the only OpenZFS implementation that has support for TRIM. TRIM support was integrated into FreeBSD in 2012 to improve the performance of pools consisting of SSDs and other flash-based devices. TRIM support provides feedback to the FTL (Flash Translation Layer) about blocks that are no longer in use and can be recycled, allowing the SSDs firmware to better manage wear leveling and garbage collection.

The OpenZFS project has work in progress to integrate a different, universal TRIM/UNMAP feature across all supported platforms; however, it is not expected to be completed until 2018. This new implementation is more advanced and is expected to further improve performance, but in the meantime, ZFS users on all other platforms are left with no TRIM support at all.

Jails

Solaris and its derivatives have zones, and advanced container technology based on the concepts pioneered by FreeBSD jails. This means that, from the beginning, ZFS had advanced integration with zones, and when ZFS was ported to FreeBSD, those features came along and were adapted to work with FreeBSD jails. Linux does not have a direct analogue to jails or zones, and so at this time has no support for ZFS in containers.

In FreeBSD, a dataset (ZFS filesystem) can be marked as “jailed.” When this flag is set, the filesystem is no longer able to be mounted on the host system (Global Zone in Solaris parlance). This is a security feature. Datasets that have been delegated to a container have mount points relative to the root of the container. A jailed dataset with a mount point of /etc that became mounted on the host system could change the root password and other configuration, which would allow a user from the container to control the contents of files that may end up on the host.

Once a filesystem has the jailed property set, the ‘zfs jail’ command can be used to delegate a dataset and all its children to a specific jail. Provided the allow.mount_zfs parameter is activated, root inside the container has complete control over the dataset and its children, except for the jailed property and limits, such as quotas. This allows root in the container to create and manage new datasets, snapshots, and all other features of ZFS. Root in the jail can even take advantage of the regular ZFS delegation feature, and further delegate access to commands and properties to regular users in that jail. This flexibility allows high-concurrency multi-tenancy with relative ease.

At ScaleEngine, we use these features to allow customers SSH access to their video storage via a jail so that they are isolated into an untrusted container with access to only their own files. User delegation and container delegation allow us to have remote replication of customer data without requiring any privileged access, and to isolate our customers while still providing them unfettered access to their own files.

Licensing[1]

When Sun Microsystems released ZFS as part of OpenSolaris, they did so under the CDDL (Common Development and Distribution License, version 1.0). The CDDL is derived from the Mozilla Public License (MPL) and attempts to hold a middle ground between the GPL (viral) and the BSD license (permissive). Code licensed under the GPL must always remain so, and any derived or combined work, in source, binary, or other form, must be licensed under the GPL. The BSD license places no restrictions on how the licensed code is used, only that the copyright notices must not be removed from the source and must be reproduced in other formats. In contrast, source code released under the CDDL must remain under the CDDL in its source form, but binaries produced from it may be licensed in any way the creator chooses, as long as the modified source is still available under the CDDL. Files licensed under the CDDL may be combined with files licensed under other licenses, whether open source or proprietary. Sun saw this as giving businesses more flexibility in how they licensed their end product, while ensuring that all ZFS features remained open source. Combining OpenZFS with FreeBSD under its liberal license means not having to fret about hidden compliance problems.

The Free Software Foundation (FSF) issued a statement on April 11, 2016, clarifying that distributing CDDL-licensed software, ZFS specifically, as part of a GPL-covered work (the Linux Kernel), is a violation of the GPL license[2]. "It is not enough to require that the combined program be free software somehow. It must be released, as a whole, under the original copyleft license (GPL)." So, while under the CDDL license, you can combine the code with GPL-licensed code, and produce a binary module licensed under the GPL, this is still not compatible with the GPL, because the combined source code must also be released under the GPL, a condition not allowed by the CDDL. Luckily, these restrictions apply to redistribution, not to private use of the code. "The GNU GPL has no substantive requirements about what you do in private; the GPL conditions apply when you make the work available to others." So, you may use ZFS, but you cannot distribute it as a complete product. The Ubuntu project sees it differently[3]: "zfs.ko, as a self-contained filesystem module, is clearly not a derivative work of the Linux kernel, but rather quite obviously a derivative work of OpenZFS and OpenSolaris." The Software Freedom Conservancy (SFC) disagrees[4]. It would seem to be prudent to avoid this legal quagmire, and just use FreeBSD, with its simple two-clause license.

> The primary design consideration for ZFS is the safety of the data. Every block that is written to the filesystem is accompanied by a checksum of the data, stored with the other metadata. That metadata block also has a checksum, as does its parent, all the way up to the top-level block, called the uber block.

Conclusions

If the integrity of your data is important, OpenZFS is the only open-source solution you can trust. The best platform for running OpenZFS is FreeBSD. Take advantage of the entire ecosystem of features, utilities, applications, and solutions that make FreeBSD a leading solution for servers and storage.

If you would like to learn more about ZFS and how to apply its advanced features to your storage challenges, pick up copies of *FreeBSD Mastery: ZFS* and *FreeBSD Mastery: Advanced ZFS* at your favorite retailer or by visiting www.zfsbook.com. ●

ALLAN JUDE is VP of operations at ScaleEngine Inc., a video streaming content distribution network, where he makes extensive use of ZFS on FreeBSD. Allan is a FreeBSD src and doc committer, and was elected to the FreeBSD core team in summer 2016. He is also the host of the weekly video podcast BSDNow.tv (with Benedict Reuschling), and coauthor of *FreeBSD Mastery: ZFS* and *FreeBSD Mastery: Advanced ZFS* with Michael W Lucas.

[1] This article is not legal advice and you cannot and should not rely on it as such.

[2] https://www.fsf.org/licensing/zfs-and-linux

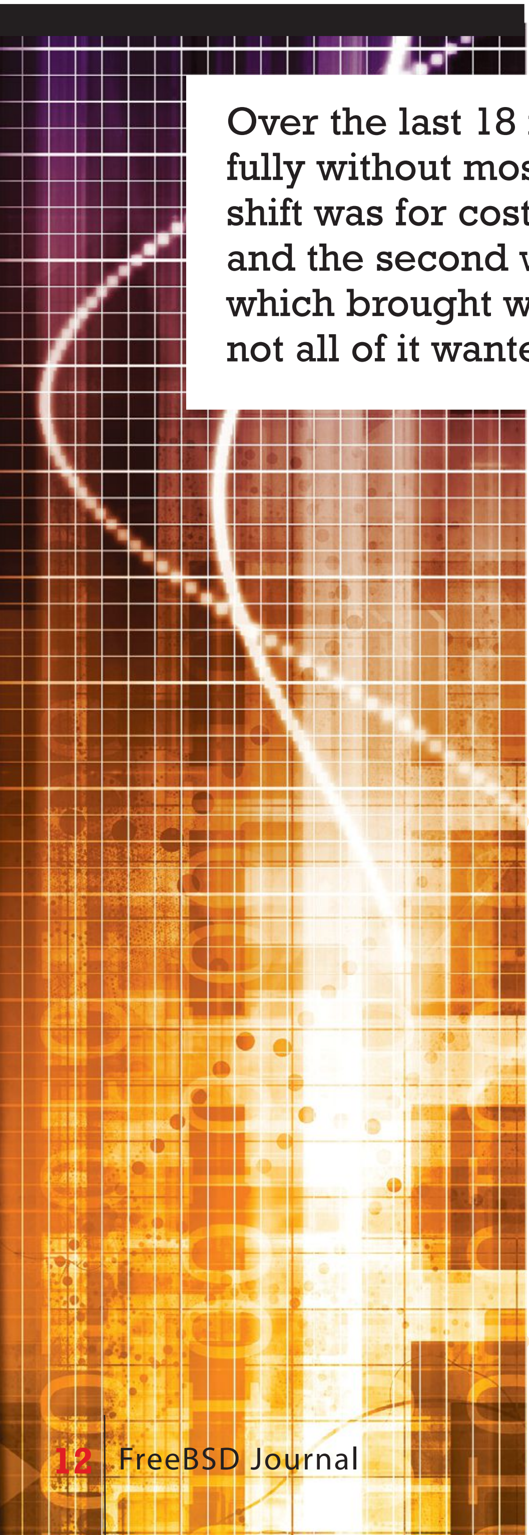
[3] https://insights.ubuntu.com/2016/02/18/zfs-licensing-and-linux/

[4] https://sfconservancy.org/blog/2016/feb/25/zfs-and-linux/

SEE TEXT ONLY

FROM Microservices to Monoliths

BY DAVE COTTLEHUBER



Over the last 18 months, we've re-platformed twice, hopefully without most people noticing at all. The first major shift was for cost advantage, changing our host provider, and the second was moving to the latest Debian release, which brought with it a flurry of patches and changes—not all of it wanted.

There was a feeling we were beholden to our OS and its patches, rather than to our customers and our business. systemd introduced a cascade of changes requiring changes to all of the services we run, and the changes continue to roll in over time.

Migrating hosting providers enabled us to move our core database servers, running Apache CouchDB (https://couchdb.apache.org/), to much larger and faster machines, using SSD and much more RAM. Building indexes is faster, which allows us to deploy code faster as well. In fact, it's now possible to cache our entire database and indexes in RAM, which has helped the responsiveness of our whole site.

We've always had a fairly microservices-like architecture, and it's stood the test of time. The front-end application is written in Perl using the Catalyst (http://www.catalystframework.org/) framework, and communicates with workers in a variety of programming languages, using RabbitMQ (http://www.rabbitmq.com/) as a message broker between services that run on several different servers. Our two main databases, Apache CouchDB and Kyoto Tycoon (http://fallabs.com/kyototycoon/), have built-in replication, which provides both application-level redundancy and also simplifies operations when doing backups or upgrades.

Despite that, microservices introduced small delays at every step—network latency due to roundtrips, further reliance on stable Internet connections, and extra conversions between JSON and the native programming languages used for each service. End-to-end testing of our application was also very complicated. Something needed to change.

The Best of Both Worlds?

We're not Twitter scale, and we don't need the feature set of Amazon or Google's clouds. We're also mindful of avoiding Cloud Lock-in. Our profitability as a business won't change much if we halve or double the infrastructure we use. We asked, is there a way to have the best of both worlds? The decoupling of microservices, without the latency? The debugging simplicity of a monolith, without the interruptions from continual upgrades? Could we have the operational flexibility of a cluster without the risk of a catastrophic meltdown? Could we have a test environment that ran on a laptop, but that still matched production?

Probably a number of you are mumbling Docker (https://www.docker.com/) under your breath. Some of you are getting sweaty palms and thinking of Linux containers (https://linuxcontainers.org/). But we'd tried them, and it wasn't the answer. We spent more time trying to get the tightly coupled container stack working together than actually shipping code to production, or improving the stability and reliability of our services.

The key failure of the container vision today is that, unless you are Google scale, where you have a significant cost advantage from reduced server footprint, and where you can afford a fleet of container infrastructure engineers to keep up with the evolving landscape, the operational effort simply doesn't stack up to deliver the benefits. It shouldn't be necessary to rewrite how we do logging, monitoring, packaging, and deployment, and to dedicate engineers to maintaining and grooming the container monster, just to simplify shipping code to production.

Plain Old-fashioned Boring Infrastructure

What we needed was some plain old-fashioned boring infrastructure. Loosely coupled at each layer, without introducing latency, and still allowing the flexibility of the container-think movement. Something that had a long support life span if we needed it, but that didn't compromise our ability to keep on the front foot for patches and security of both the OS and the apps that came with it.

In the end, we settled on three core changes:

- move our OS from Debian Linux to FreeBSD
- switch out distributed container-style VM microservices to paired physical servers
- migrate our Perl-based Catalyst core app over to Elixir and Phoenix

Many of you will know of these already, but here's a glimpse into our thinking.

FreeBSD

FreeBSD (https://www.freebsd.org/) is one of the original Free UNIX-like operating systems, and is going stronger than ever. It powers Netflix's mighty streaming servers, and is used in a modified form in Sony Playstations and Apple's iOS and OSX. Most Internet providers use FreeBSD in some form. FreeBSD also has a long support life span for the core OS while at the same time allowing us to use the latest ports and packages—a neat mix of backward compatibility.

And to be honest, FreeBSD already had a leg-up as a couple of us have been using it for a while.

The three big advantages of FreeBSD for us are zfs (http://open-zfs.org/wiki/Main_Page), jails (https://www.freebsd.org/doc/handbook/jails.html), and ports (https://www.freebsd.org/ports/references.html).

zfs is arguably the leading filesystem of all time, with great flexibility, and power including inbuilt high-speed compression, data checksums (no bitrot or silent data corruption), snapshots for replication and backup. It also supports boot environments (http://callfortesting.org/bhyve-boot-environments/), which is a clever snapshot-based way of managing upgrades safely. This makes testing and ultimately deploying a new version of FreeBSD, or our apps and services deployed upon it, largely risk-free, and very, very simple.

FreeBSD jails are about a decade old, and similar to Linux containers conceptually. However, there are no venture-backed companies fighting over turf in the hope of achieving a VMWare-like monopoly, and the software is well integrated into the operating system and community tooling. The performance of jailed applications is effectively the same as that of running in the main kernel, both from a network and a filesystem standpoint.

The ports tree is a significant feature that all BSD-derived operating systems share a massive repository (subversion or git as you please) of every piece of open-source software you could possibly imagine. As our core business is providing simplified domain purchase and management through custom software, we are often in the position of needing a specific version of a tool or application, or needing custom patches deployed immediately while we wait for the upstream application owners to merge a patch, or for it to trickle down into the OS distribution's package manager. With the ports tree, we have a custom private repo for our own packages, and the ability to carry and patches or to hold back specific versions for our own needs, in a very simple and straightforward fashion. Where we've needed new packages, or to get changes committed, it's proved extremely simple to do so. This makes maintaining our own infrastructure very simple indeed—installing a handful of packages gets us up and running much faster than in the past.

There is a fourth advantage for us, however. The FreeBSD community is close-knit, with a reasonably consistent culture about doing things right. In practice, this means there's very little gap between issues we identify or knowledge we are missing, and the developers and community creating it. The documentation on FreeBSD itself is part of this culture of doing things right, and the integration between the OS itself and the tools it ships with are the result, and we are already looking forwards to contributing further to the community.

Overall, as a result of moving to FreeBSD we hope to spend significantly less effort managing our infrastructure, and more time invested in improving our services and business.

A Pair of Servers

FreeBSD's jails allow us to run microservices on a single box. Arguably this is no different from Linux containers in practice; however, we are not fighting to accommodate a stream of changes that add no value to our business along the way, and we are not forced to change out our operations tools and processes. The value to our customers is not in having the latest container tech running; it's in having simple and reliable services for the infrequent times they need to acquire or manage their domains throughout the year.

In each region, we are deploying a pair of physical servers, each one providing the same services and applications. Within each pair, we are using DNS round-robin name resolution and CARP, a low-level IP availability solution built into FreeBSD, to provide load balancing and failover at a network level between our boxes.

The next layer up in the paired server stack is the awesome haproxy (http://www.haproxy.org/) load balancer, which we use to ensure that we direct our users to the closest and best-performing application server. haproxy also allows us to dynamically remove and add back-end services from the pool, whether during deployment or maintenance, and communicates across the cluster to maintain a transparent view of services for our customers.

This consolidation brings disparate virtual machines back onto the same server, while still using FreeBSD jails to maintain the microservices-like separation. Luckily, none of our apps have required major changes to make them run on FreeBSD—UNIX standardization has been a huge benefit here. When this is complete, we'll have significantly reduced our app latency by removing the network round-trips that we have today.

The key difference here from the Docker-style container architecture is that there is very little coordination or dependency between these layers.

CARP is fundamentally a network protocol, and we could easily disable it should there be any issues, or choose some other facility. haproxy could be replaced by nginx (http://nginx.org/) which we already use today for slightly different functionality. zfs provides an incredible filesystem, available within jails and to the core operating system, as a solid and reliable platform for our services and your data. Logging, monitoring, and upgrades are all done using the same decoupled tools using well-known UNIX standards in place for decades—and there's nothing wrong with preserving that simplicity where it suits us.

Elixir and Phoenix

Since almost the beginning of iwantmyname, the programming language Erlang/OTP (http://www.erlang.org/) has been at the heart of things. It's the programming language that Apache CouchDB is developed in, as well as RabbitMQ, our message broker, and our core search (https://iwantmyname.com/?domain=) application is also written in Erlang. Its robustness has shown time and time again as it transparently deals with issues such as transient connection failures to our API partners, and it has generally required significantly less maintenance effort than our other services. As a concurrent functional language with soft-real-time characteristics, it is ideally suited to building websites and services that make heavy use of asynchronous internal and external APIs.

Our front-end app, written in Perl's Catalyst (http://www.catalystframework.org/) framework, was leading edge when we first started using it, but it's become more and more of a hindrance in evolving to a more robust, mobile first system. Perl's forking worker model means that we use a significant amount of RAM across our infrastructure just to ensure we can handle what is definitely not "web scale" user and network load.

After experiencing several years of solid Erlang reliability, we picked Elixir (https://elixir-lang.org/), a new functional programming language that runs on the same Erlang VM, and the Phoenix (http://phoenixframework.org/) web framework written in the same language, to upgrade and eventually replace our front-end application.

Erlang's BEAM virtual machine provides Elixir, and Phoenix, with a screamingly fast, robust, and reliable concurrent web framework, without the memory overhead of a forking model, that is able to handle many concurrent connections transparently. Both Erlang and FreeBSD are used heavily by WhatsApp, and they shot to worldwide notice (https://duckduckgo.com/html/?q=erlang+whatsapp) when Facebook acquired the low-staffed app.

Live Debugging

Sometimes stuff breaks in production for no apparent reason, and we need to know why—in a hurry. In recent months, we traced and debugged transient Internet outages, upstream API changes, timeouts and failures, unanticipated third-party library concurrency model changes, and much much more—much pain and frustration!

Debugging was a painful process, usually involving reading all the log files, using low-level Linux tools like strace, all the while inserting print statements and redeploying furiously while trying to understand the underlying issues from the resulting flood of information.

Aside from being able to roll back safely any changes using boot environments (http://callfortesting.org/bhyve-boot-environments/) and packages, our new stack provides some incredible introspective live debugging capabilities, which makes it easier for us to deal with problems in real-time and without downtime, and most importantly without needing to change compiler settings or edit our production code on the fly. These advantages alone would be reason enough to move. The Erlang VM provides a native erlang tracing (http://erlang.org/doc/man/dbg.html) library, and the community has extended this with erlang dtrace (http://erlang.org/doc/apps/runtime_tools/DTRACE.html) support, and the delightful recon (https://ferd.github.io/recon/recon_trace.html). FreeBSD itself supports natively DTrace (http://dtrace.org/blogs/), the powerhouse introspection tool first developed for Sun Solaris, and since ported to several other platforms.

We've dubbed this setup the "FRECK Stack"—FreeBSD, RabbitMQ, Elixir, CouchDB, Kyoto Tycoon. Yes, we had some closely related abbreviations in mind, but we managed to control our childish mirth.

Looking Ahead

At the end of the day, iwantmyname (https://iwantmyname.com/) is a domain registrar, and when it comes to domains, a "plain old boring infrastructure" is exactly what we—and you—should want.

With FRECKS, we're now more stable and more secure than we've ever been—allowing us to truly focus our brain muscles on overdue UI and UX upgrades. More changes to our iwantmyname platform are coming, but behind the scenes; it'll hopefully be smooth sailing out front—touch wood! •

DAVE COTTLEHUBER enjoys building and supporting distributed systems, especially when he can use both Elixir and FreeBSD. He shaves yaks at https://iwantmyname.com/, a domain reseller based in New Zealand. He lives in Vienna, Austria, with his wife and three boys, next to a lake. In his free time, he drinks Zwickl, eats schnitzel, wears lederhosen, and contributes to a number of open-source projects.

SEE TEXT ONLY

A Comparison of Unix Sandboxing Techniques

Why sandboxing is different from historic approaches to Unix security, how we got where we are, and how Capsicum compares with Linux's seccomp(2) and OpenBSD's pledge(2).

BY JONATHAN ANDERSON

Today's users need more protection than traditional Unix systems have been able to deliver. The authors of operating systems have traditionally had a great deal of interest in systemic notions of privilege (e.g., the authority to inject code into the kernel via modules), but the users of computing systems often require finer-grained models of access control (e.g., the ability to share a single contact or delegate management of a single calendar). Rather than protecting multiple users from each other, the operating systems of today's end-user devices must protect single users from their applications and those applications from each other. Historic Unix-derived systems have not made this task easy; in some cases, the protection users need has not even been possible.

Protection was a first-class objective of early, general-purpose operating systems and the hardware on which they ran [Lamp69, And72, SS72]. This early focus led naturally to the exploration and design of rigorous, general-purpose protection primitives such as capabilities [DV66] and virtual memory [BCD69, Lamp69]. In the transition from Multics to Unix dominance, this focus was lost. The result was a highly portable operating system that would go on to dominate contemporary thinking about operating systems, but with security features primarily organized around one threat model: users attacking other users (including accidental damage done by buggy software under development). This security model—Discretionary

This work has been sponsored by the Research and Development Corporation of Newfoundland and Labrador (contract 5404.1822.101) and the NSERC Discovery program (RGPIN-2015-06048), the Defense Advanced Research Projects Agency (DARPA), and the Air Force Research Laboratory (AFRL) (contract FA8650-15-C-7558).

Access Control (DAC)—can be implemented with Unix owner/group/other permissions or with Access Control Lists (ACLs), but it does not provide adequate support for application sandboxing. FreeBSD, Linux, and MacOS eventually acquired frameworks for enforcing systemic security policies such as multi-level security and integrity enforcement [WFMV03, WCM+02, Wat13], collectively known as Mandatory Access Control (MAC). Such policies represent the interests of system owners and administrators and provide an additional dimension along which enforcement can be specified, but they are better-suited to tasks such as protecting high-integrity files from low-integrity data than to supporting sandboxing in unprivileged applications.

The goal of sandboxing is to protect users from their own applications when those applications are exposed to untrusted content. Complex applications are regularly exposed to content from malicious sources, often embedded within difficult-to-parse protocols and file formats. This is especially true on the Internet, where even the most basic use cases involve ASN.1 parsing (for TLS) as well as parsing documents, Web pages, images, and videos as well as interpreting scripts or encoding/decoding cookies. Even the humble **`file`**(1) command was patched in 2014 for vulnerabilities in its parsing code [SA14:16]. Once a process is compromised by malicious content, the goal of a sandboxing policy is to limit the potential for damage to a small set of known outputs. For example, a compromised word processor may be able to corrupt its output files, but it should not be able to search through a user's home directory for private keys or credit card details. Sandboxing-specific features (or, as they are sometimes referred to, attack-surface–reduction features) such as FreeBSD's Capsicum, OpenBSD's **`pledge`**(2), and Linux's **`seccomp`**(2) have appeared comparatively recently; we compare their effectiveness below.

Sandboxing with DAC/MAC

Prior to the introduction of sandboxing features in commodity operating systems, valiant efforts were made to confine or sandbox applications with the tools that were available. These efforts met with varying degrees of success, depending on how well the designed security policy fit onto a discretionary or mandatory access control (DAC or MAC) model. The most successful applications of sandboxing required relatively small code changes to meet their security objectives, which tended to fit the coarse-grained security model of DAC or the system-security perspective of MAC. Less successful forays into sandboxing required thousands of lines of code, sometimes with large amounts of hand-crafted assembly, a requirement for system (superuser) privilege and—often—a failure to truly enforce the desired security policy.

An early—and relatively successful—implementation of sandboxing was Provos et al's privilege separation of the OpenSSH server [PFH03]. This work used discretionary access control features to prevent a compromised SSH server process from exercising the privileges of the superuser. The SSH server requires superuser privilege in order to bind to TCP port 22, but it is desirable that a compromised SSH process not be able to access system resources such as the filesystem before a user has authenticated (i.e., the server should be put into a pre-auth sandbox); it is also desirable that the server post-authentication only be able to exercise the authority granted to the authenticated user (from within a post-auth sandbox). Provos et al split the SSH server process into a trusted monitor process that retained superuser privilege and untrusted child processes that would use the superuser privilege to drop privilege, changing their user and group IDs to those of unprivileged users. In the pre-auth sandbox, an SSH server process could run as the **`nobody`** user and have its root directory changed to an empty directory using the **`chroot`**(2) system call. In the post-auth sandbox, a process would have its UID/GID changed to those of the authenticated user. This approach to sandboxing was successful for two reasons:

1. User-oriented policy: The goal of SSH privilege separation is to keep compromised processes from exercising the authority of the superuser. This policy goal aligns well with the Unix DAC model: it can be expressed entirely in terms of UIDs, GIDs, and filesystem directories with Unix permissions. The policy does not protect a user from misbehavior post-authentication: it protects the system and other users.

2. Extant privilege: Operations such as changing a process's UID or root directory require superuser privilege, which the SSH

> server undergoing privilege separation already possessed. The **sshd** process was already a security-critical piece of software run as **root**: privilege separation was a monotonic decrease in authority. This is not the case for the more general case of sandboxing, however: it is undesirable to require unprivileged software to run as **root** in order to drop privilege.

At the other end of the spectrum, we have previously compared several DAC- and MAC-based approaches to sandboxing renderer processes in the Chromium web browser [WALK10]. In that work, we found that DAC and MAC mechanisms were a poor fit for the application compartmentalization use case. DAC is designed to protect users from each other, but in the case of a Web browser—or any other sophisticated, multiprocess user application—the security goal is to limit the damage that can be done by a rogue process after it is compromised by untrusted content. DAC alone cannot control access to unlabeled objects such as System V shared memory (in Linux) or FAT filesystems (in Windows). As with OpenSSH, **chroot**(2) can be used to put a process into an environment of limited filesystem access, but unlike OpenSSH, the superuser privilege required to use **chroot**(2) is not naturally found in Web browsers (or office suites, music players, other desktop applications, etc.). Thus, in order to avail of the DAC-based protection that did exist, portions of the application had to be shipped with the **setuid** bit set on a **root**-owned binary![1]

Mandatory Access Control (MAC) is also a poor fit for application sandboxing. It requires a dual coding of policy: once in the code that describes what the application does and once in a separate policy that describes what the application is allowed to do. The SELinux policy from our original Capsicum comparison involved thousands of lines of policy, but even modern AppArmor profiles encoded in a domain-specific language can require hundreds of lines of subtle and complex policy, not including policy elements included from system policy libraries (e.g., `#include <abstractions/ubuntu-browsers.d/java>`). It can be very difficult to write and maintain complex MAC policies, with failures in functionality (e.g., a lack of access to `@{PROC}/[0-9]*/oom_score_adj`) being more obvious to

[1] Today's Chrome no longer uses the DAC-based sandbox on Linux, but the above comments about **chroot**(2) and privilege still apply to the new sandboxing model.

developers than lapses in protection (e.g., allowing access to `/usr/bin/xdg-settings`, which relies on the `PATH` environment variable not being hijacked). This complexity is a hint that we are attempting to fix the square peg of sandboxing into the round hole of MAC.

"Sandboxing" with System-call Interposition

Before comparing today's approaches to sandboxing, it is essential to understand an intermediate approach that was attempted in the late 1990s and early 2000s. This approach was attractively simple, but ultimately failed to provide the security benefits it advertised. Those who fail to learn from this now-discredited approach are condemned to repeat its mistakes in their "new" approaches to application sandboxing.

A seminal attempt to generalize policy enforcement for arbitrary applications without system privilege was Fraser, Badger and Feldman's Generic Software Wrappers [FBF00], which inspired systems such as Provos's `systrace` [Prov03]. These system-call interposition systems used userspace wrappers or shallow modifications to the system-call layer of an OS kernel to intercept system calls. Once intercepted, these calls' arguments could be inspected and a policy decision could be made as to whether or not the call should be allowed. For example, instead of using `chroot`(2) to limit a process's access to the filesystem, every `open`(2) call could be inspected and the filename argument could be compared against a whitelist of paths the process was allowed to open. System call policies could be described in languages that, while requiring dual coding as in MAC, had the benefits of concision and comprehensibility, as shown in Figure 1. System call wrappers had the twin benefits of being relatively simple to implement and relatively simple to use. Unfortunately, their simplicity translated into a failure to engage with the complexities of concurrent accesses in operating systems, as demonstrated by Watson in 2007 [Wat07].

Objects named by Unix system calls are concurrent on multiple levels. At the shallowest level, all of a process's threads are contained within the same virtual address space and can thus manipulate the same data—this includes strings being passed as arguments to system calls. When system call wrappers work in userspace, a malicious process can submit a system call for execution with a path that is known to be whitelisted and then, while the wrapper's policy check is executing, modify the value of the memory containing the filename to a different path. Thus, the path that is checked against the policy can be different from the path that is eventually accessed. To use Watson's language, this is a Time-of-check-to-time-of-use (TOCTTOU) vulnerability [Wat07].

TOCTTOU vulnerabilities are not merely found at this shallow layer of interception, however. If they were, interception would only need to be done via RPC to be secure. System call wrappers are vulnerable in a deeper, more fundamental way: even if the name used to reach an OS object such as a file remains constant, the meaning of that name can change. Path lookup is an incremental operation in Unix: looking up a file named `/home/jon/foo.txt` will involve interactions with at least four vnodes in a virtual filesystem (the root node, two directories and the file itself). While each individual lookup (e.g., retrieving the `jon` directory entry) must be done with due care for

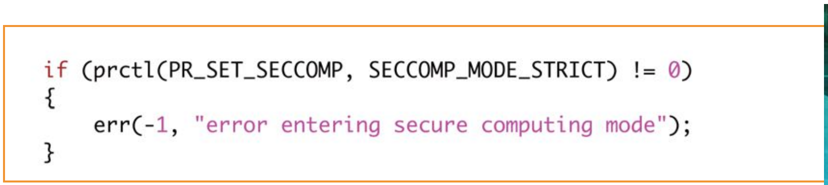
```
# GSWTK policy
#include "../../wr.include/platform.ch"

wrapper bsd_noadmin {
        bsd::op{mount || unmount || ...} pre {
                return WR_DENY | WR_BADPERM;
        };
}

# systrace policy
Policy: /usr/sbin/named, Emulation: native
        native-__sysctl: permit
        native-accept: permit
        native-bind: sockaddr match "inet-*:53" then permit
        native-break: permit
        native-chdir: filename eq "/" then permit
        native-chdir: filename eq "/namedb" then permit
        native-chroot: filename eq "/var/named" then permit
        native-close: permit
        native-connect: sockaddr eq "/dev/log" then permit
        ...
```

Fig. 1
Policies governing system call wrapper behavior for the Generic Software Wrapper Toolkit and `systrace` (reproduced from [FBF00] and [Prov03], respectively).

concurrency, e.g., while holding a lock, the overarching path lookup is not an atomic operation. A path is a list of instructions, not a name. While one process is walking a directory hierarchy, another can be changing the filesystem, moving files, moving directories, even changing symbolic links. System-call interposition, even if performed via RPC with no possibility of in-memory path substitution, cannot guarantee that the file named by a path at the time a policy decision was made is the same file that will be looked up by the system call doing the lookup.



```
if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT) != 0)
{
    err(-1, "error entering secure computing mode");
}
```

Fig. 2 Entering the original, "pure" version of Linux's secure computing mode is trivial. Once a process is in `seccomp` mode it can never leave.

Fundamentally, the weakness of system-call interposition is that its policy decisions (i.e., checks) are not made atomically with the effects of those decisions. This is not a vulnerability that requires a patch, but a fundamental limitation of the approach; it is why such methods are no longer used on contemporary operating systems (OpenBSD expunged `systrace` in April of last year [Gros16]). However, even though system-call interposition systems have been deprecated, the underlying concept returns to haunt more modern sandboxing frameworks.

A Comparison of Sandboxing Frameworks

More recently, open-source Unix derivatives have implemented new frameworks to aid in application sandboxing. These frameworks include, most comparably, Linux's `seccomp`(2), OpenBSD's `pledge`(2), and FreeBSD's Capsicum (`capsicum`(4))[2]. Although they were all created with the goal of enabling simple sandboxing, they have achieved varying degrees of success.

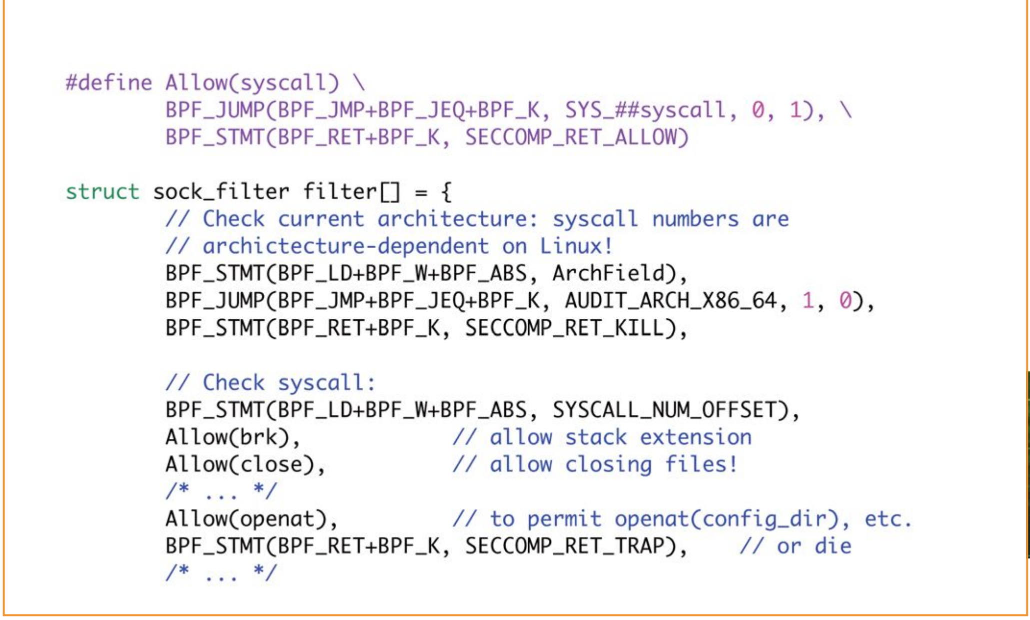
Linux: `seccomp`(2)

Since 2005, Linux has included a feature called "secure computing mode," or `seccomp`(2) for short [Cor09]. The original version of `seccomp`(2) provided a strong, comprehensible security policy: processes in "secure computing mode" can use the `read`(2) and `write`(2) system calls to operate on files they have previously opened (or had delegated to them), `sigreturn`(2) to support signal delivery, and the `exit`(2) system call to terminate the process. It is simple for a process to enter `seccomp` mode, as shown (in abridged form[3]) in Figure 2. This policy had the benefit of clarity and it did permit processes to operate as filters performing otherwise-pure computation, but very few applications are able to perform meaningful work within such a restrictive sandbox. For example, we previously found that Chrome's use of the "pure" `seccomp`(2) mode required over a thousand lines of security-critical assembly-language code to forward system calls outside of the sandboxed process and into a trusted process that would perform the system calls on its behalf [WALK10].

To provide a richer environment for computing, modern `seccomp`(2) allows programs to specify their own security policy beyond the four system calls enumerated above. In this new version of `seccomp`(2), a process can specify a program to check each system call's validity before executing it. This program is written in the BPF bytecode format. The BSD Packet Filter (BPF) [MV93], inspired by the CMU/Stanford Packet Filter (CSPF), itself inspired by earlier work on the Xerox Alto [MRA87], is a virtual machine that interprets bytecode. It was originally designed to facilitate high-performance networking by allowing userspace processes to describe a filter for the kernel to apply to network packets without giving up the safety of the kernel/user mode separation. When applied to `seccomp`(2), BPF provides a syntax for describing programs that check system calls within the Linux system call handler.

An example of a simple system-call whitelisting filter is shown in Figure 3. This illustrates the extreme flexibility and programmability of `seccomp-bpf`: almost any check that can be imagined on a system call's arguments can be expressed in an assembly-like language like BPF. However, the corollary to this is that because anything can be checked by the programmer,

[2] Discussion of Apple's Sandbox framework and its MAC Framework underpinnings is left to other sources [Wat13].
[3] Full source code for the examples in this section can be found at https://github.com/trombonehero/sandbox-examples.



```
#define Allow(syscall) \
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_##syscall, 0, 1), \
        BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW)

struct sock_filter filter[] = {
        // Check current architecture: syscall numbers are
        // archictecture-dependent on Linux!
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, ArchField),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, AUDIT_ARCH_X86_64, 1, 0),
        BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),

        // Check syscall:
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, SYSCALL_NUM_OFFSET),
        Allow(brk),             // allow stack extension
        Allow(close),           // allow closing files!
        /* ... */
        Allow(openat),          // to permit openat(config_dir), etc.
        BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRAP),    // or die
        /* ... */
```

Fig. 3
An example of a simple **seccomp-bpf** filter that allows the **brk**(2), **close**(2), and **openat**(2) system calls to proceed (based on an example from Bernstein [Bern17]).

everything must be checked by the programmer. In order to build a meaningful whitelist of system calls, not only must the offset of the syscall number within a larger structure be exposed to user-mode programs, the provided filter must also inspect the current architecture in order to interpret the syscall number (Linux uses different system call numbers on different architectures). Furthermore, as semantics are left to the programmer, it is possible—indeed, all too easy—to construct inconsistent system call policies that deny some operations while allowing equivalent operations to be performed. For example, the policy in Figure 3 does not allow unrestricted **open**(2) calls, but it permits **openat**(2), which can be made to behave equivalently to **open**(2). A **seccomp-bpf** filter is intimately tied to the details of the program whose behavior it filters, making it the responsibility of the application authors, but constructing a **seccomp-bpf** system call filter requires meticulous attention to the sorts of details (assembly programming in BPF opcodes, layouts and semantics of Linux kernel syscall handling structures) that are entirely outside of most application authors' experience and working knowledge.

Beyond simple syscall whitelists, **seccomp-bpf** is both more complex and more problematic. It is possible to construct **seccomp-bpf** filters on system call arguments such as filenames, but as with GSWTK and **systrace**, it is impossible to check paths meaningfully at the system-call handling layer. A program may be permitted to access **/var/tmp/***, but if **/var/tmp/foo** is a symbolic link that can be updated in a race with the BPF filter, what policy has truly been enforced? The **openat** example at https://github.com/trombonehero/sandbox-examples demonstrates how a process restricted using **seccomp-bpf** can escape from its intended bounds, in this case creating files outside of an application's intended working directory.

For all of these reasons, **seccomp-bpf** alone is insufficient to truly sandbox arbitrary application code. A complete application sandbox must also use the Linux **clone**(2) system call to sequester a process within a new IPC namespace (to cut off access to the host's global System V IPC namespace), network namespace (interfaces, routing, firewall, **/proc** and **/sys/class/net**, etc.), mount namespace (similar to **chroot**(2)), and PID namespace (to cut off inappropriate uses of **kill**(2)). Creating such namespaces requires the **CAP_SYS_ADMIN** privilege, which is effectively equivalent to superuser privilege on Linux[4]. Thus, creating an effective application sandbox on Linux requires running programs as **root** or creating **setuid** binaries.

OpenBSD: pledge(2)

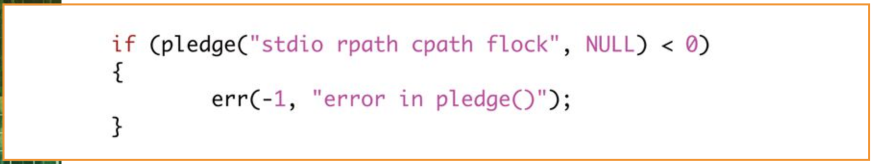
Since v5.9 was released in 2016, OpenBSD has shipped with **pledge**(2), a mechanism for putting a process into a "restricted-service operating mode"[5]. The manual page for **pledge**(2) does not describe it as a security mechanism [Pled17], but other communications by its developers do [deRa15]. The essence of **pledge**(2) is a simpler,

[4] The POSIX.1e draft standard [Pos1e] specified fine-grained superuser privileges called "capabilities" such as **CAP_NET_RAW**, **CAP_SETGID** or **CAP_SYS_ADMIN** as decompositions of traditional superuser privilege. These "capabilities" are different from the traditional computer science definition of capabilities [DV66], which are discussed below. The POSIX.1e draft was withdrawn and is not in force, but portions of it have been implemented by various operating systems (e.g., FreeBSD's audit implementation and Linux's "capability" framework).

[5] The previous **tame**(2) mechanism was introduced in v5.8 but not enabled by default.

more easily used take on the seccomp(2) concept. Instead of defining a BPF program to filter out system calls, pledge(2) groups system calls into categories such as stdio (which includes read(2), write(2), dup(2), and clock_getres(2)) and rpath (which allows read-only filesystem effects from chdir(2), openat(2), etc.). It is possible to make a pledge with the empty string, in which case no further system calls but _exit(2) are permitted, but this can result in processes aborting when atexit(3) code triggered by C startup routines in _start() call mprotect(2) on libc.

pledge(2) is considerably simpler to use than equivalent seccomp-bpf functionality. Figure 4 shows an example of pledge(2) use that applies a system call filter to the current process using more system calls than that of Figure 3. However, as with seccomp-bpf, this simple, superficial filtering of system calls provides illusive security guarantees. The provided system call categories may usefully describe the requirements of trivial OpenBSD base system applications, but for complex applications, categories such as wpath are effectively meaningless. If an application needs to open private files for writing, then wpath must be "pledged," but wpath also authorizes opening any file on the filesystem with the correct DAC mode for writing. Unlike seccomp-bpf, pledge(2) makes policy construction simple, but like its Linux analog, it makes the construction of inconsistent or meaningless policies easy to do by default.



```
if (pledge("stdio rpath cpath flock", NULL) < 0)
{
        err(-1, "error in pledge()");
}
```

Fig. 4 A system call filtering policy is considerably simpler to install with pledge(2) than with seccomp-bpf.

The pledge(2) system call also takes a paths argument containing a whitelist of allowable paths, but that functionality has been marked as "unavailable" in the pledge(2) manual page since early 2016 [Pled17]. Were it available, the shallow whitelisting functionality would suffer from the same TOCTTOU vulnerabilities as systrace and seccomp-bpf. However, the greatest weakness of pledge(2) is that a compromised process can disable the security mechanism if the original pledge(2) call included the exec system-call category. Despite claims that "abilities can never be regained" [Pled17] and "in OpenBSD, once a mitigation is working well, it cannot be disabled" [deRa15], pledge(2) does not have the one-way property of seccomp(2) or capsicum(4). In those systems, a process that enters a restricted state remains there together with all of its subsequently-created children, but an OpenBSD process's pledge(2)-restricted state is cleared on exec(2).

As with seccomp-bpf (and GSWTK/systrace before that), system call filtering with pledge(2) is insufficient to apply a meaningful security policy to applications more complex than read–compute–write filters. The difference is that, although it is the simpler framework to use, pledge(2) is not backed by clone(2)-based mechanisms for implementing more rigorous security policies. Thus, as with the now-discontinued-by-OpenBSD systrace, pledge(2) should be seen as a debugging and mitigation feature to catch unskilled adversaries rather than a rigorous mechanism on which to build security policies.

FreeBSD: capsicum(4)

The Capsicum compartmentalization framework is different from seccomp-bpf and pledge(2) in two key ways. First, Capsicum employs a principled, coherent model for restrictions on processes when applications are compartmentalized. This is implemented by Capsicum's capability mode. Second, Capsicum employs fine-grained, monotonic reduction of authority on specific OS objects accessed via attenuated file descriptors, called capabilities.

Capability Mode

Like seccomp-bpf and pledge(2), capsicum(4) supports putting processes into a restricted mode in which system calls behave differently from "normal" processes. The key distinction is how the restrictions are chosen. Rather than a superficial focus on specific system calls, many of which have overlapping responsibilities and provide independent means of accomplishing the same objective, Capsicum focuses on a fundamental principle underlying them all: access to global namespaces.

In Capsicum, the cap_enter(2) system call causes a process to enter capability mode, in which all access to OS objects (files, sockets, processes, shared memory, etc.) must be done through capabilities (described below) rather than using ambient authority. Ambient authority describes the normal authority of a process to act on behalf of its user, doing anything that the user is permitted to do by the Unix DAC model. This includes access to other processes via PID, files via path or NFS file handle, sockets via protocol addresses, shared memory via System V IPC name or POSIX shared memory path, etc. By contrast, a process in capability mode is not allowed to access any new resources via global namespaces (path, PID, protocol address, etc.). Resources represented by already-open file descriptors (or descriptors passed into a process via Unix message passing) are normally subject to restrictions described below (under "capabilities"). New file descriptors may also be derived from existing descriptors using system calls such as accept(2) or even openat(2), provided that only local names are used. In the case of openat(2), this requires that path search start relative to an already-open directory descriptor, not AT_FDCWD, and that path evaluation only traverse "down" inside a directory and not "up" via "..". This restriction on path lookup is enforced within FreeBSD's namei() function, deep within the kernel and atomic with the lookup being policed.

The policy enforced by Capsicum's capability mode is internally consistent, as it is based on a fundamental principle rather than shallow system call syntax. It can enforce the same restrictions as the limited, internally-consistent use cases of seccomp-bpf and pledge(2): if a process enters capability mode with no resources held but readable/writable file capabilities, no side effects can be caused on the system except those described by the descriptors. To enable more sophisticated behaviors, Capsicum provides capabilities to facilitate principled sharing of resources within a coherent security model.

Capabilities

The historic concept of a capability in computer science is that of an identifier for an object combined with operations that can be performed on it. This sense of the word was described by Dennis and Van Horn in the late 1960s [DV66], and its echoes can be heard in Unix today. In Dennis and Van Horn's conception, a capability was an index into a list of capabilities maintained by the supervisor on behalf of a process. This concept carried forward into Multics and then morphed into the file descriptor as we know it today in Unix [RT78].

> “The policy enforced by Capsicum's capability mode is internally consistent, as it is based on a fundamental principle rather than shallow system-call syntax.”

Like capabilities, file descriptors are indices into a supervisor-maintained list of OS objects; they are also associated with operations that may be performed on them based on the flags they were opened with (e.g., O_RDONLY). Unlike capabilities, however, file descriptors carry unexpected, implicit authority with them that cannot be monotonically reduced. For example, an application cannot open(2) a descriptor with flag O_RDWR, dup(2) it, commute the new descriptor to a read-only descriptor, and share it with an untrusted worker process. Even when a file descriptor is opened read-only, the Unix DAC model will still permit system calls like fchmod(2) to—perhaps unexpectedly—manipulate file metadata.

Capsicum's implementation of capabilities provides for the monotonic reduction of fine-grained rights ("authorities") on specific objects. It does this by attaching rights such as CAP_READ, CAP_FSTAT, CAP_MMAP, CAP_FCHMOD, etc., to descriptors. These classes of behaviors correspond to methods on kernel objects and are related to sets of system calls that require them. For example, to open a read-write file relative to a directory with openat(2), that directory descriptor must have at least CAP_READ, CAP_WRITE, and CAP_LOOKUP enabled for it. Outside of capability mode, unsandboxed processes using ambient authority with system calls such as open(2) are returned file descriptors with all rights implicitly granted. This preserves compatibility with traditional Unix semantics while allowing for uniform enforcement of capability rights both inside and outside capability mode. Rights on descriptors can be attenuated using cap_rights_limit(2), descriptors can be inherited by or passed to sandboxed processes, and new descriptors derived

```
 if (lpc_bootrom())
        fwctl_init();

+#ifndef WITHOUT_CAPSICUM
+caph_cache_catpages();
+
+if (caph_limit_stdout() == -1 || caph_limit_stderr() == -1)
+       errx(EX_OSERR, "Unable to apply rights for sandbox");
+
+if (cap_enter() == -1 && errno != ENOSYS)
+       errx(EX_OSERR, "cap_enter() failed");
+#endif

 /*
  * Change the proc title to include the VM name.
  */
 setproctitle("%s", vmname);
```

Fig. 5 Only minimal code changes were required to add Capsicum support to the `bhyve` hypervisor. `caph_cache_catpages`() pre-opens a directory, `caph_limit_std{out,err}`() limits the rights held on stdout and `stderr`, and `cap_enter`() enters capability mode.

from existing ones (e.g., via `accept`(2) or `openat`(2)) derive their rights from their parent objects. This allows delegation with confidence.

Sandboxing with Capsicum

Capsicum allows application authors to apply rigorous security policy to their applications with—in some cases—a minimum of effort. Today, even moderately complex applications such as hypervisors and Web browsers can support rich use cases by opening resources (including resource-bearing resources such as directories and server sockets), limiting the rights associated with those resources and then entering capability mode. The work required to sandbox the bhyve hypervisor in this way is shown in Figure 5. Efforts are ongoing to make the Capsicum model applicable to broader classes of applications, including applications that require access to external resources such as powerboxes [Yee04], even when they are oblivious to sandboxing features [AGW17].

Starting from a rigorous foundation, Capsicum is a platform that can support complex behaviors. Since its security policies are both simple and coherent, application authors can build supporting services on this foundation without requiring expertise in kernel internals or the fear of constructing an incoherent security policy. We therefore see Capsicum as a generative platform that enables application authors to focus on what they do best, using rigorous security-enabling tools without requiring extreme security expertise. It is our hope that providing authors with tools for safe software construction will enable future applications to better protect users, not just from each other, but from their own applications. ●

References

[And72] Anderson, J. P. "*Computer Security Techology Planning Study*", ESD-TR-73-51, Electronic Systems Division, US Air Force, 1972, URL: https://csrc.nist.gov/publications/history/ande72.pdf.

[AGW17] Anderson, J.; Godfrey, S.; and Watson, R. N. M. "*Toward Oblivious Sandboxing with Capsicum*", *FreeBSD Journal*, July/August 2017, URL: https://www.freebsdfoundation.org/past-issues/security.

[Bern17] Bernstein, O. "*Denying Syscalls with Seccomp*", Eigenstate, retrieved August 2017, URL: https://eigenstate.org/notes/seccomp.html.

[BCD69] Bensoussan, A.; Clingen, C.; and Daley, R. "*The multics virtual memory*", in SOSP '69: Proceedings of the Second Symposium on Operating Systems Principles, 1969, pp. 30–42, DOI: 10.1145/961053.961069.

[Cor09] Corbet, J. "*Seccomp and sandboxing*", LWN.net, 2009, URL: http://lwn.net/Articles/332974.

[Cor12] Corbet, J. "*Yet another new approach to seccomp*", LWN.net, 2012, URL: http://lwn.net/Articles/475043.

[deRa15] de Raadt, T. "`pledge`(): *a new mitigation mechanism*", 2015, accessed September 2017, URL: http://www.openbsd.org/papers/hackfest2015-pledge.

[DV66] Dennis, J. and Van Horn, E. "*Programming semantics for multiprogrammed computations*", *Communications of the ACM 9(3)*, 1996, pp. 143–155, DOI: 10.1145/365230.365252.

[FBF00] Fraser, T.; Badger, L.; and Feldman, M. "*Hardening COTS software with generic software wrappers*", in Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (DISCEX), 2000, DOI: 10.1109/DISCEX.2000.821530.

[Gros16] Grosse, J. "`systrace`(1) *is removed for OpenBSD 6.0*", 2016, URL: http://daemonforums.org/showthread.php?t=9795.

[Lamp69] Lampson, B. "*Dynamic protection structures*", in AFIPS '69 (Fall): Proceedings of the AFIPS 1969 Fall Joint Computer Conference, 1969, DOI: 10.1145/1478559.1478563.

[MRA87] Mogul, J. C.; Rashid, R. F.; and Accetta, M. "*The Packet Filter: An Efficient Mechanism for User-level Network Code*", in Proceedings of the 11th Symposium on Operating Systems Principles (SOSP), 1987, pp. 39–51, URL: https://dl.acm.org/ft_gateway.cfm?id=37505.

[MV93] McCanne, S. and Jacobson, V. "*The BSD Packet Filter: A New Architecture for User-level Packet Capture*", in Proceedings of the USENIX Winter 1993 Conference, 1993, URL: https://www.usenix.org/legacy/publications/library/proceedings/sd93/mccanne.pdf.

References continue next page

References continued

[Pled17] "`pledge`—*restrict system operations*", in OpenBSD System Calls Manual, 2016–17, retrieved September 2017, URL:https://man.openbsd.org/pledge.2.

[Pos1e] Portable Applications Standards Committee of the IEEE Computer Society, "*Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment #: Protection, Audit and Control Interfaces [C Language]*", IEEE Draft Standard (withdrawn), 1997, URL: http://wt.tuxomania.net/publications/posix.1e/download.html.

[Prov03] Provos, N. "*Improving Host Security with System Call Policies*", in Proceedings of the 12th USENIX Security Symposium, 2003, URL: https://dl.acm.org/citation.cfm?id=1251371.

[PFH03] Provos, N.; Friedl, M.; and Honeyman, P. "*Preventing Privilege Escalation*", in Proceedings of the 12th USENIX Security Symposium, 2003, pp. 231–242, URL: https://www.usenix.org/legacy/events/sec03/tech/provos_et_al.html.

[RT78] Ritchie, D. and Thompson, K. "*The UNIX time-sharing System*", in Bell System Technical Journal 57(6), 1978, pp. 1905-1929, DOI: 10.1002/j.1538-7305. 1978.tb02136. x.

[SA14:16] FreeBSD. "*Multiple vulnerabilities in* `file`*(1) and* `libmagic`*(3)*", 2016, URL: https://www.freebsd.org/security/advisories/FreeBSD-SA-14:16.file.asc.

[SS72] Schroeder, M. D. and Saltzer, J. H. "*A Hardware Architecture for Implementing Protection Rings*", in Communications of the ACM 15(3), 1972, pp. 157–170, DOI: 10.1145/361268.361275.

[Wat07] Watson, R. N. M. "*Exploiting concurrency vulnerabilities in system call wrappers*", in Proceedings of the 2007 USENIX Workshop on Offensive Technologies (WOOT), 2007, URL: http://static.usenix.org/event/woot07/tech/full_papers/watson/watson.pdf.

[Wat13] Watson, R. N. M. "*A Decade of OS Access-Control Extensibility*", in Communications of the ACM 56(2), 2013, pp. 52–63, DOI: 10.1145/2408776.2408792.

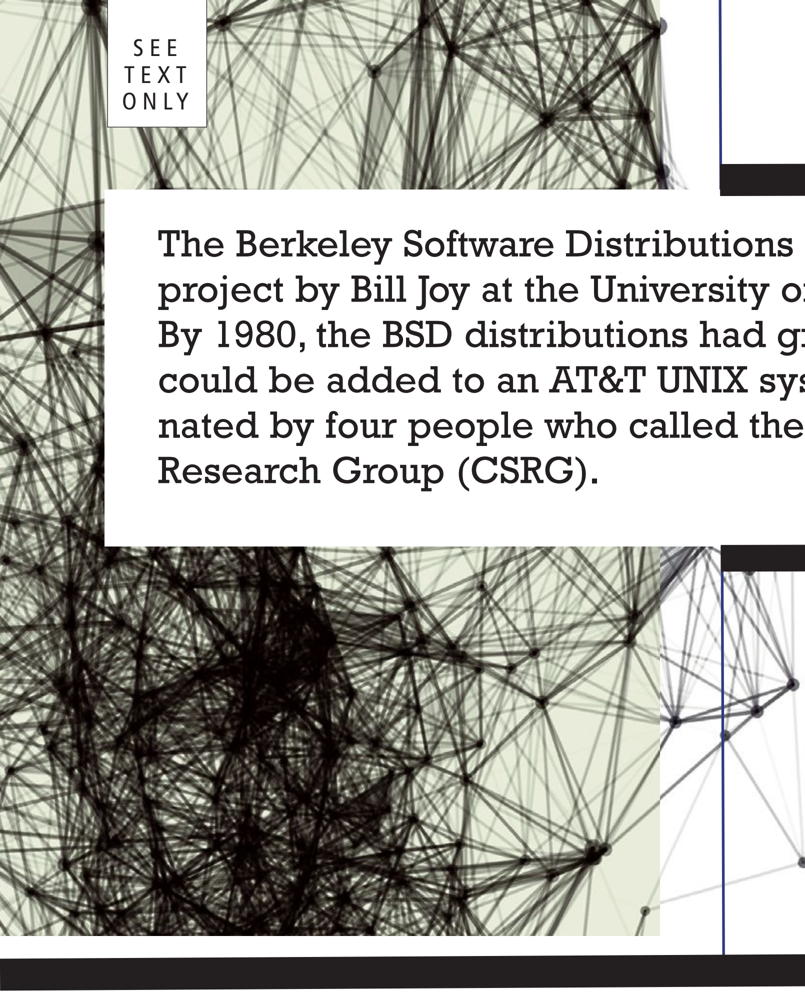
[WALK10] Watson, R. N. M.; Anderson, J.; Laurie, B.; and Kennaway, K. "*Capsicum: practical capabilities for UNIX*", in Proceedings of the 19th USENIX Security Symposium, 2010, URL: https://www.usenix.org/legacy/events/sec10/tech/full_papers/Watson.pdf.

[WCM+02] Write, C.; Cowan, C.; Morris, J. et al. "*Linux Security Modules: General Security Support for the Linux Kernel*", in Proceedings of the 11th USENIX Security Symposium, 2002, URL: https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright.pdf.

[WFMV03] Watson, R.; Feldman, B.; Migus, A.; and Vance, C. "*Design and implementation of the Trusted BSD MAC framework*", in Proceedings of the 2003 DARPA Information Survivability Conference and Exposition (DISCEX '03), 2003, DOI: 10.1109/DISCEX.2003.1194871.

[Yee04] Yee, K. "*Aligning security and usability*", IEEE Security and Privacy 2(5), 2004, DOI: 10.1109/MSP.2004.64.

JONATHAN ANDERSON is an Assistant Professor in Memorial University of Newfoundland's Department of Electrical and Computer Engineering, where he works at the intersection of operating systems, security, and software tools such as compilers. He is a FreeBSD committer and is always looking for new graduate students with similar interests.



SEE TEXT ONLY

The Berkeley Software Distributions (BSD) was started as a one-man project by Bill Joy at the University of California at Berkeley in 1977. By 1980, the BSD distributions had grown from a few programs that could be added to an AT&T UNIX system to a complete system coordinated by four people who called themselves the Computer Systems Research Group (CSRG).

The Evolution Of FreeBSD Governance

BY MARSHALL KIRK MCKUSICK AND BENNO RICE

At that time, BSD development began being managed using SCCS, the first source-code control system. By 1983, the socket interface had been designed, and TCP/IP implemented underneath it allowing a small set of trusted external contributors to log into the CSRG development machines over the ARPAnet (which later became the Internet) and directly update the sources using SCCS. The CSRG staff could then use SCCS to track changes and verify them before doing distributions. This structure formed the basis for the current BSD-based projects once BSD was spun off from the university as open source.

The Formation of the FreeBSD Project

The final release from Berkeley was an open-source version of BSD called Networking Release 2 which was later rereleased as 4.4BSD-Lite. The release was missing six kernel files that still contained AT&T proprietary code. Bill Jolitz wrote replacements for these six missing files and released a system called 386BSD that ran on the commodity PC hardware. Frustrated with the slow development pace of Bill's 386BSD work, a set of developers forked 386BSD to start the FreeBSD Project.

Following the CSRG model, the FreeBSD Project used a CVS source-code repository to manage the code. The only distribution method for their early releases was agonizingly slow 14.4K dialup modems. As the FreeBSD 1.0 release approached, they were looking for a way to more quickly reach a larger audience. Jordan Hubbard approached Walnut Creek CD-ROM whose

business model was creating CD-ROM distributions of open-source software that they could then sell. The FreeBSD developers' goal of expanding FreeBSD's distribution matched well with Walnut Creek CD-ROMs business model, so Walnut Creek CD-ROM was happy to pick up FreeBSD as one of their distributions. For its part, Walnut Creek CD-ROM provided the high-powered development machines needed by the FreeBSD developers to host the CVS repository and to manage the release engineering of distributions that Walnut Creek CD-ROM sold. As the popularity of FreeBSD grew, Walnut Creek CD-ROM hired several of the FreeBSD developers to work on FreeBSD full-time.

As the use of FreeBSD expanded, so did the number of software packages included with the distributions. To keep the size of the FreeBSD distribution from exploding, the ports collection was created.

The base FreeBSD system has just the critical programs and libraries. The ports collection (which currently has over 25,000 packages) can then be used to supplement the base system with the programs needed to complete a system's functionality. So, a desktop machine installs a window manager, web browser, and a mail client from the ports collection. A machine providing a web server installs a program like Apache from ports.

Initially, everyone working on FreeBSD could commit to the CVS repository, but as the number of developers involved grew, that became untenable. With the move to Walnut Creek CD-ROM, a core team was created to make the commits and to decide who else should be able to commit to the repository. GNATS was brought up to do bug tracking.

The FreeBSD Project Moves into Companies

As FreeBSD expanded in size and needs, and became core technology at more companies, the source code repository and main development machines moved from Walnut Creek CD-ROM to Yahoo, whose entire company ran on FreeBSD. Realizing that having the project dependent on the munificence of a single company was undesirable, Justin Gibbs created the FreeBSD Foundation in 2000 in the hope that it could eventually garner enough support to fully support the project infrastructure.

It took a decade before the FreeBSD Foundation was fully able to support the project resources. Today it provides many things, including staff to head the marketing and release engineering teams; a staff person to oversee both developers receiving grants from the foundation; and other foundation staff working on projects that tackle needed parts of the system development that volunteers do not have the time to do or are not interested in doing.

Initially, the core team was permanently appointed, eventually approaching nearly 20 members. By 2000, only about a third were consistently active in the project while another third were not participating at all. This deadwood caused the business of the core team to grind almost to a halt. There was growing frustration among the committers that decisions were not being made in a timely manner and/or that the core team members acted abruptly and radically in ways that others felt bordered on impunity.

As a result, some key developers, both in and out of core, increasingly took matters into their own hands. Nobody wanted to relinquish their perceived prestigious core-title voluntarily. To gain better accountability, promote faster decision making, and to have a natural mechanism that cleared out the deadwood, a group of central developers proposed letting the developers elect the core team.

Warner Losh together with Poul-Henning Kamp, Wes Peters, and others drew up a set of bylaws that created the current structure where the core team is nominated from and elected by committers every two years. The underlying philosophy was that if core cannot be trusted, the project is doomed, but at the same time you cannot legislate common sense. Creating core by electing a bunch of random people into the role seemed unlikely to bring about project unity or even consensus on important matters. But it was agreed that democracies are the worst form of government except for all the others. So, an elected core was deemed to be the best solution.

With some arm-twisting, the original core team adopted the bylaws thus bringing in the first elected core team of nine members. Unsurprisingly, only a few of the original core members were carried over to the first elected core. The net effect was generally agreed to be that there was little change in the effectiveness of the core team. However, overall contentment of the developers improved as it was a lot harder to argue with the implicit authority of an elected body that appears to have the support of a majority of the electorate.

The core team is tasked with keeping the FreeBSD Project running. It approves new committers, resolves differences between committers, and manages committer discipline using such mechanisms as suspension of commit privileges. They also handle any other top-level issues that arise within the project.

To streamline management of various areas, the core team has created other teams or responsible people (referred to as "hats") who own certain aspects of the project. These teams include:

- The port manager team that oversees the 217 ports committers who maintain the ports tree.
- The documentation team that oversees the 126 documentation committers who develop the FreeBSD documentation and prod other committers if their documentation needs updating.
- The Security Officer, along with a team that handles security issues and oversees the release of security alerts and updates.
- The seven-member system administration team that maintains the FreeBSD infrastructure.
- The release engineering team that consists of Glen Barber and about 10 other committers who assist him with FreeBSD releases.
- The quality assurance team that runs continuous integration builds and creates an ever-growing set of regression tests. Additionally, the FreeBSD Foundation assists with advocacy and marketing. The group is headed by Anne Dickison who works with members of the FreeBSD community to provide promotion, outreach, and social networking for FreeBSD.

The FreeBSD Project Today

Project collaboration was initially handled using a single mailing list. Over time the traffic on this list grew until it became necessary to split it out into multiple lists, each focused on a given topic area such as networking, file systems, ports, documentation, announcements, and general questions. Eventually, the proliferation of mailing lists made it difficult to deal with issues that spanned several areas. Some collaboration happened via bug tracking, initially in GNATS and later in Bugzilla, but these tools were lacking when it came to reviewing larger changes, particularly if involving developers outside of FreeBSD.

In 2014 an instance of Phabricator, an open-source collaboration tool written and released by Facebook, was set up to allow detailed pre-commit review of larger changes. Phabricator facilitates detailed review and discussion of a proposed change somewhat similar to GitHub "pull requests." Phabricator has created an easier venue for non-committers to propose changes to FreeBSD as they are able to create their own Phabricator accounts, post their recommended changes, and have Phabricator automatically suggest reviewers or otherwise notify appropriate FreeBSD developers that a change needs review.

FreeBSD Source-Code Control

When the FreeBSD project began, the founders chose to use the CVS source-control system. With the release of newer source code management tools like Subversion in 2000, the pressure to move to a more modern tool began to increase. This pressure only increased with the release of the newer wave of tools such as Git and Mercurial. The branching models and commit atomicity of all of these tools were highly attractive along with the fact that all of them were generally easier to deal with than CVS. Eventually, after a lot of discussion, test conversions, and verification, the project moved to Subversion with the rationale that it was fairly close in operation to the CVS system, and that Git and Mercurial could both function on top of it if needed.

Despite the conversion to Subversion, discussion has continued about moving to something newer. There are many FreeBSD developers actively using Git, and with the advent of GitHub and its pull request model, there are ongoing discussions on whether FreeBSD should adopt GitHub or something like it. The FreeBSD Project has a presence on GitHub but it is purely read-only. Pull requests and issues opened on GitHub can only be addressed by having them moved over to Phabricator or Bugzilla before being committed into Subversion.

FreeBSD Workflow

As the project grew, more formal structures were needed to ensure smooth workflow without inhibiting innovation. Outside developers produce bug fixes and updates to FreeBSD. Using mailing lists or consulting the source-code-control logs, they identify an appropriate committer with whom to work to get their changes incorporated into FreeBSD. Another option is to create a Phabricator account to raise an issue and have the Phabricator infrastructure identify the appropriate committer or group with whom to work.

Committers (of which there are currently 371) are authorized to commit changes to specific parts of the system. These system parts are broken into three main committer groups: documentation, ports, and source. Many committers are in more than one group. Committers normally work in a self-defined subset of the groups in which they are a member. All changes (other than trivial ones) require review by at least one other committer.

Historically, committers could simply make any changes they saw fit. This policy led to broken infrastructure, especially when changes were made to systems that were more central, such as the virtual memory subsystem. To combat these problems, developers were encouraged to seek review of their changes before committing. These changes were formalized by requiring tags to be added to commit log messages indicating:

- which other project member had reviewed the changes,
- the sponsoring organization (e.g., the project member's employer),
- the bug report number from which it was identified,
- the Phabricator thread on which it was discussed,
- when to send a reminder to merge the change to older stable/release branches, and,
- if appropriate, an acknowledgement that the commit fixes an earlier mistake made by the committer (the "Pointy Hat" tag).

Guidelines on How to Work and Play Together

Though the project had long had a set of guidelines on how members should interact with each other, it did not have explicit rules and procedures to be followed when the guidelines were violated. Rules and procedures were added in response to some developer disagreements and misbehavior, and detailed clear behavioral expectations. These initial rules were primarily based around interactions within CVS.

The initial rules did not approach the clarity of behavioral expectation contained in a more modern Code of Conduct. The initial rules did provide examples of the types of sanctions that the core team could impose in response to breaches of those rules. The initial rules sufficed until 2015 when there were some serious cases of project member misbehavior that spread beyond the bounds of the project itself. This event led to efforts to augment the initial rules with a modern Code of Conduct and a set of attendant processes to deal with issues like this event in the future.

FreeBSD Recruitment

As with all successful open-source projects, developers lose interest or have insufficient time and leave the project. To avoid deadwood, it is important to have metrics to gauge when developers have left. The FreeBSD Project uses the metric of one year without doing a commit to drop an individual's commit privileges.

To keep the project viable, new developers must be recruited and brought into the project. There are several ways new contributors come into contact with the project. One is simply by discovering the project and becoming involved directly. Another is by coming into contact via a university or college course. A third is by working at a company that uses FreeBSD in its products or services. FreeBSD also takes part in the Google Summer of Code and has gained many contributions through this program.

To provide a welcoming and easily entered community, it is important to make the project visible and to provide new developers with mentors to help them learn the policies and procedures used by the project. Committers working with active developers can nominate them to be brought into the project as a new committer. Core is responsible for deciding whether to admit new committers. To ease the transition and to ensure that new committers understand the culture, procedures, and rules of the project, they are assigned a mentor (usually the person that nominated them) to review their changes and ensure that they get proper external review. Once they have gotten up to speed, typically in six to twelve months, their mentor deems them ready to work independently.

FreeBSD Development Model

The project has been quite good at identifying areas for change. When the changes are small and/or contained to a small area of the project, architecting and developing those new areas has gone smoothly. However large or highly impactful changes have been difficult. Two examples stand out: the shift from a single-threaded to a multithreaded kernel and the move from CVS to

Subversion.

The shift away from CVS was first suggested in 1999. It was recommended that the project move to BitKeeper, which was then in public beta. This suggestion did not pan out, but it, and subsequent discussions around other tools, all fit a general pattern where someone would suggest moving to one tool, others would object and/or suggest moving to other tools, yet others would either vocally support or object to one or more of the previous suggestions, and in the end the discussion would die out with no real conclusion. In 2008, Peter Wemm, in his role as both a member of the cluster administration team and as one of the CVS repository managers, cleared the deadlock by dint of simply picking Subversion, doing all the work necessary to perform and validate the migration, and making it all happen. Subversion was chosen due to its close match to the semantics of CVS, its relative maturity, and the ability of the two other most commonly cited competitors, Git and Mercurial, to interoperate with Subversion.

Moving the kernel from single-threaded to multithreaded was a similarly large task, but, in this case, the problem was one of personalities and disagreements over architecture. Berkeley Software Design Inc. (BSDi) purchased Walnut Creek CDROM in 2000. This purchase provided developers employed by Walnut Creek access to the source code of BSD/OS, BSDi's commercial BSD derivative. One of these developers, John Baldwin, used ideas from BSD/OS and Solaris to create an architecture for a multithreaded FreeBSD kernel. Another developer, Matthew Dillon, preferred a different architecture that was similar to the approach taken in the Amiga kernel. Significant conflicts arose between these two over how the multithreading project should continue. These conflicts were exacerbated by the core team not wanting to actively pick sides in a technical debate. In the end, due to other reasons, Matthew's commit access was removed and he left the project to found the DragonFlyBSD project.

To try to avoid situations like these two, core introduced the "FreeBSD Community Process", a more formalized mechanism for proposing and deciding on important or contentious changes within the project. The idea is to avoid discussions degenerating into an interminable argument on the mailing lists with ultimately no action being taken.

The FreeBSD Community Process is modeled on similar ideas in other projects, particularly the Python Enhancement Process (https://www.python.org/dev/peps/pep-0001/), the Joyent RFD Process (https://github.com/joyent/rfd/blob/master/README.md), and even the venerable IETF RFC Process (https://www.ietf.org/about/standards-process.html).

Committers who want to make a change that will result in a nontrivial effect on the FreeBSD user base, or retrospectively, anyone having backed out a change after running into contention over something that turned out less trivial than they anticipated, writes down what they propose to change. Their proposal describes the problem they are trying to solve, outlines how they propose to solve it, and indicates any consequential impacts the proposal may have. After being vetted by a FreeBSD Community Process editor, the document is added to the FreeBSD Community Process index, committed into the FreeBSD Community Process repository, and published for discussion. Each FreeBSD Community Process proposal is a living document and can be updated to reflect any conclusions resulting during the discussion.

Once consensus has been achieved, or the discussion has gone on for enough time, the core team votes on accepting the proposal. The core team is expected to vote according to the mood of the discussion around the proposal.

FreeBSD Core Team Interaction with the FreeBSD Committers

Historically, the core agenda was private, and all communications within the core team, whether by email, IRC, or the monthly video conference, were kept private. Communication of the core team's activities to the committers was limited to a monthly report on the actions that they had taken that was prepared by the core secretary. The core secretary was not a member of core but managed core's agenda and handled many of the communications with core. The report consisted of a brief summary of the actions taken by core and discussed only actions that had already been taken. Because of the monthly report's retrospective perspective, there was little opportunity for committers to participate in core's deliberations.

After years of prodding by the committers, the core team recently began working to be more transparent and to provide an opportunity for committers to have more input to core's deliberations. The core team started providing its agenda to developers before their meetings. Core is also looking into allowing committers to attend their video-conference meetings. Attending video-conference meetings will require identifying agenda items that require core-only deliberations that may need careful handling such as disputes between developers.

FreeBSD Security Team

The role of the Security Officer has evolved over the years. Initially it was simply a title for the person tasked with looking after security-related issues for the project. In 2002, an official charter (https://www.freebsd.org/security/charter.html) was adopted that also acknowledged that there was more work than one person could handle and that there would be a Security Team that reported to the Security Officer. In the last few years it has also become apparent that finding someone with the background and time to be the Security Officer is no small feat. In response to this expansion of responsibility, the core team has recast the Security Officer role to be more of a managerial one with the Security Team acting as a pool of people who can do work as needed to address security issues, draft advisories, and ensure patches get into all the relevant branches.

Summary and Conclusions

Like many open-source projects, FreeBSD started out with the Benevolent Dictator(s) For Life governance model. While this approach can be effective, it often leads to stagnation and an aging out and/or burning out of those in charge. For these reasons, FreeBSD moved first to a core team and later to an elected core team.

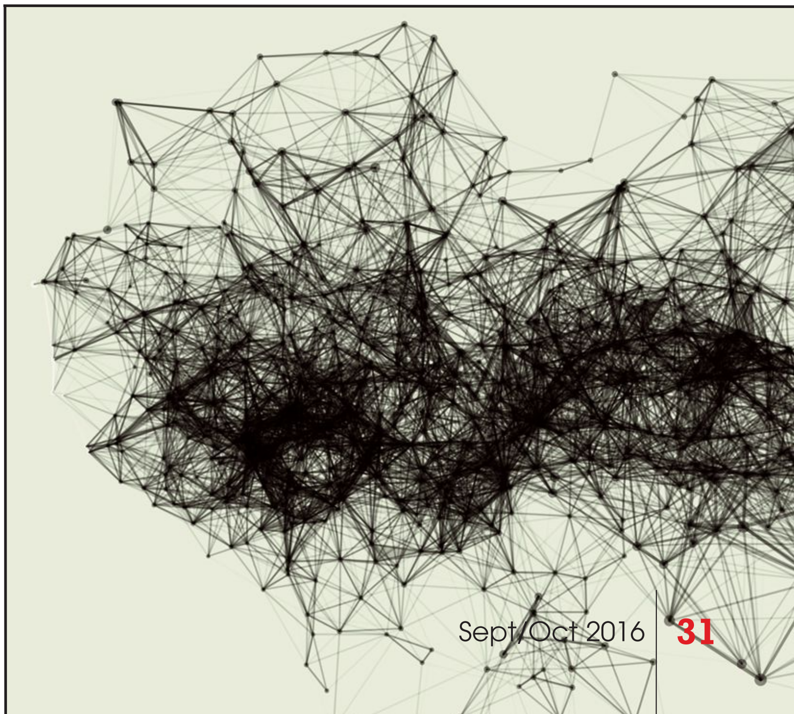
In its 24-year history there have been four major changes in leadership, each of which has been beneficial and allowed the project to move forward and tackle new problems. It has also helped the project avoid aging out; the median age of the committers has consistently remained in the mid to high 30s. Young enough to have the time and energy to push projects forward, but old enough to have the necessary wisdom to avoid rat holes, and with the patience and experience to avoid technical debt by doing things right rather than settling for a quick hack.

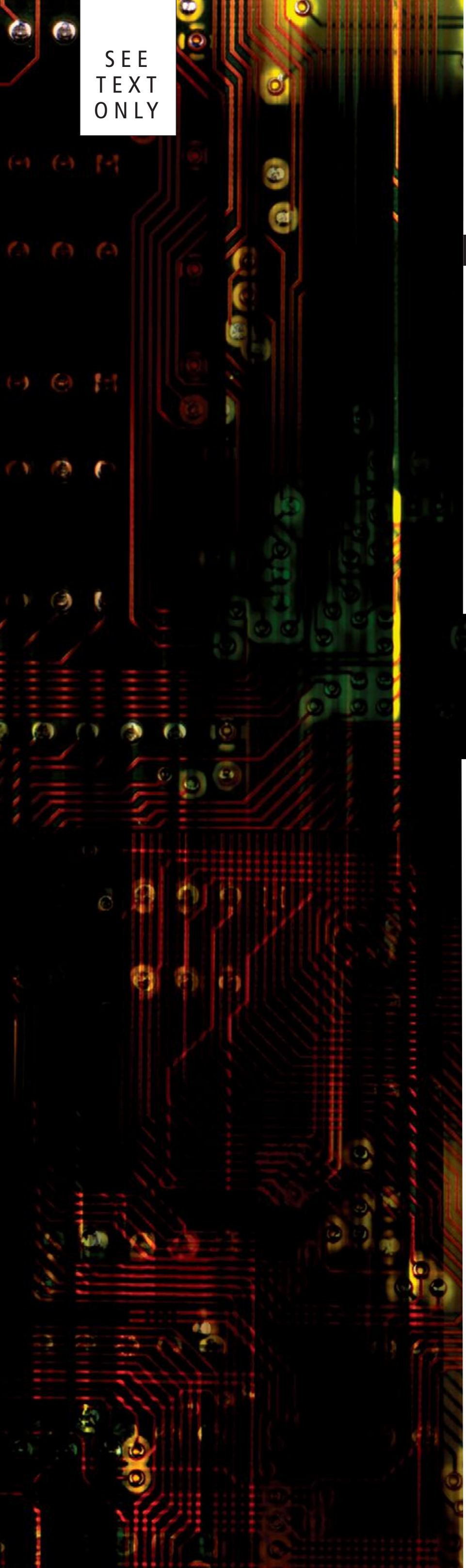
The creation of the FreeBSD Foundation has also been important in ensuring that the FreeBSD Project has the resources that it needs to be successful. Importantly, the FreeBSD Foundation has recognized that its role in the FreeBSD ecosystem is to let the core team and the committers determine the technical direction of the FreeBSD Project while supporting it with infrastructure, marketing, and outreach.

Governance is mundane yet critical to the success of an open-source project. Too much and the project becomes stifled. Too little and the project can go off the rails either as a success disaster or by getting bogged down from lack of enough structure to get things done. To date, the FreeBSD Project has gotten governance right, but keeping it humming requires constant tuning. ●

DR. MARSHALL KIRK MCKUSICK writes books and articles, teaches classes on UNIX- and BSD-related subjects, and provides expert-witness testimony on software patent, trade secret, and copyright issues particularly those related to operating systems and filesystems. He has been a developer and commiter to the FreeBSD Project since its founding in 1993. While at the University of California at Berkeley, he implemented the 4.2BSD fast filesystem and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD

BENNO RICE is a software engineer for Dell EMC's Isilon division and also a committer and core team member for FreeBSD.





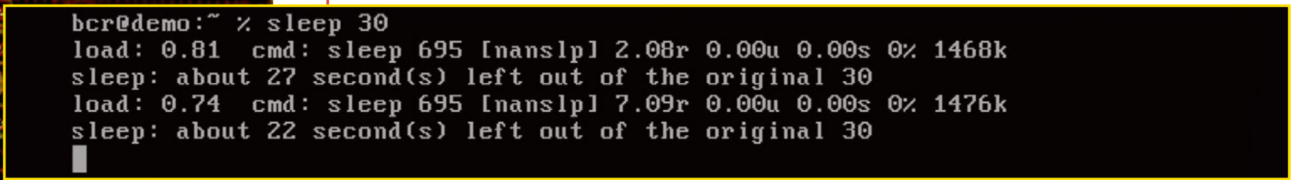
BY BENEDICT REUSCHLING

Linux, and a couple other Unix Systems that do not implement SIGINFO STAY IN THE DARK!

Every now and then in a sysadmin's daily activities, there comes a time when a command takes a little longer before it gives back the prompt. This may be due to many causes. Maybe there is a process that is either waiting for input from another source or is busy processing something and has nothing to print to the console in the meantime. The longer there is no output on the screen, the more nervous the user becomes and starts to wonder whether the process is still active or is progressing at all. In those situations, users logged into a BSD system are fortunate to have SIGINFO available. By pressing CTRL+T, many utilities print out additional information that would normally be left hidden.

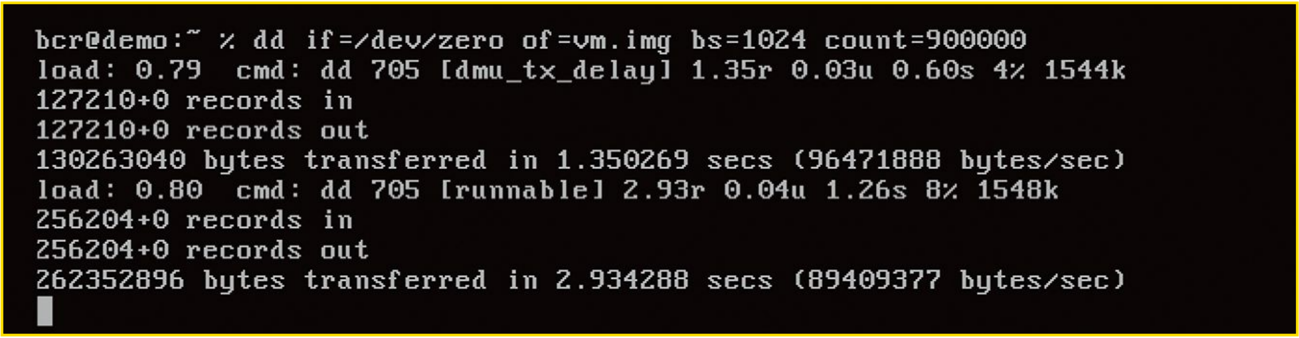
As the name suggests, SIGINFO is a signal that can be sent to a process like CTRL+Z (SIGTSTP), to stop the process, or SIGKILL to terminate a hung process. Not every operating system implements SIGINFO, as it is not part of the POSIX standard. A look at signal(3) reveals that FreeBSD implements SIGINFO. Many sysadmins on those systems have come to rely on it for that extra bit of information. Linux and a couple other Unix systems that do not implement SIGINFO stay in the dark, as nothing will be returned no matter how often CTRL+T is pressed.

The clearest example to demonstrate this is probably the sleep(1) command. The following example shows the number of remaining seconds by hitting CTRL+T two times in short intervals:



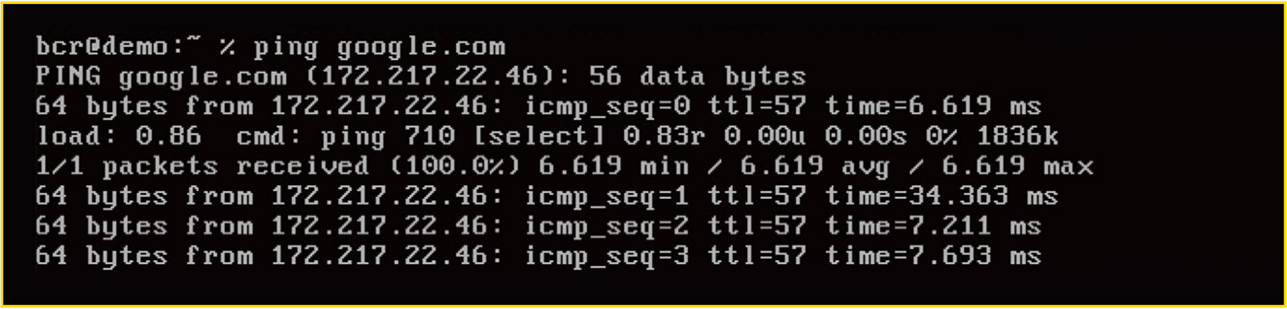
```
bcr@demo:~ % sleep 30
load: 0.81  cmd: sleep 695 [nanslp] 2.08r 0.00u 0.00s 0% 1468k
sleep: about 27 second(s) left out of the original 30
load: 0.74  cmd: sleep 695 [nanslp] 7.09r 0.00u 0.00s 0% 1476k
sleep: about 22 second(s) left out of the original 30
```

The dd command is another example of a process that can run for a long time. In this case, we create the base disk image for a virtual machine and fill it from /dev/zero. SIGINFO returns a short report and shows how many records have already been written. (next page)



```
bcr@demo:~ % dd if=/dev/zero of=vm.img bs=1024 count=900000
load: 0.79  cmd: dd 705 [dmu_tx_delay] 1.35r 0.03u 0.60s 4% 1544k
127210+0 records in
127210+0 records out
130263040 bytes transferred in 1.350269 secs (96471888 bytes/sec)
load: 0.80  cmd: dd 705 [runnable] 2.93r 0.04u 1.26s 8% 1548k
256204+0 records in
256204+0 records out
262352896 bytes transferred in 2.934288 secs (89409377 bytes/sec)
█
```

Network utilities like ping(8) and tcpdump(1) will list the received or captured packets respectively.



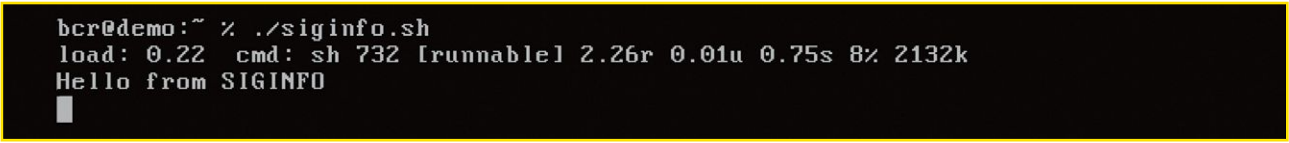
```
bcr@demo:~ % ping google.com
PING google.com (172.217.22.46): 56 data bytes
64 bytes from 172.217.22.46: icmp_seq=0 ttl=57 time=6.619 ms
load: 0.86  cmd: ping 710 [select] 0.83r 0.00u 0.00s 0% 1836k
1/1 packets received (100.0%) 6.619 min / 6.619 avg / 6.619 max
64 bytes from 172.217.22.46: icmp_seq=1 ttl=57 time=34.363 ms
64 bytes from 172.217.22.46: icmp_seq=2 ttl=57 time=7.211 ms
64 bytes from 172.217.22.46: icmp_seq=3 ttl=57 time=7.693 ms
```

To implement the SIGINFO functionality in shell scripts, a signal handler routine must be implemented that calls a function which executes when the signal is received. Here is a simple Bourne shell script that echoes “Hello from SIGINFO“.

```
#/bin/sh
info() {
    echo “Hello from SIGINFO“
}

trap info SIGINFO
while (true); do
done
```

The output looks like this:



```
bcr@demo:~ % ./siginfo.sh
load: 0.22  cmd: sh 732 [runnable] 2.26r 0.01u 0.75s 8% 2132k
Hello from SIGINFO
█
```

By default, the shell handles the CTRL+T already by echoing some details about the running process. The information from our own info() function is also displayed. Why not try out SIGINFO on many of the base system utilities that FreeBSD has installed and see if SIGINFO is implemented? A good amount of information is just a keycombination away. Soon, that habit will become a dearly missed functionality when sitting in front of an OS that does not have SIGINFO. ●

BENEDICT REUSCHLING joined the FreeBSD Project in 2009. After receiving his full documentation commit bit in 2010, he actively began mentoring other people to become FreeBSD committers. He is a proctor for the BSD Certification Group and joined the FreeBSD Foundation in 2015, where he is currently serving as vice president. Benedict has a Master of Science degree in Computer Science and is teaching a UNIX for software developers class at the University of Applied Sciences, Darmstadt, Germany.

svn UPDATE

by Steven Kreuzer

Since the topic of this issue is FreeBSD vs. Linux, I thought it would be interesting to take a closer look at some of the changes made to the LinuxKBI subsystem over the past few months. The project was originally announced in May 2016 as a way to no longer have to port changes from the Linux KMS and DRM drivers over to FreeBSD. The idea is to add a set of shims that can act as a compatibility layer to allow for these drivers to work with minimal changes, making it easier to follow upstream development and greatly reducing the diff between FreeBSD code and the original code from Linux. In addition, it also speeds up the integration of new changes in FreeBSD, which I am sure is a welcome addition to anyone running FreeBSD on bleeding-edge hardware.

Properly implement `poll_wait()` in the LinuxKPI. This prevents direct use of the `linux_poll_wakeup()` function from unsafe contexts, which can lead to use-after-free issues.
https://svnweb.freebsd.org/changeset/base/323349

Resolve IPv6 scope ID issues when using `ip6_find_dev()` in the LinuxKPI.
https://svnweb.freebsd.org/changeset/base/323351

Add more sanity checks to `linux_fget()` in the LinuxKPI. This prevents returning pointers to file descriptors that were not created by the LinuxKPI. https://svnweb.freebsd.org/changeset/base/323347

Remove unsafe access to the LinuxKPI file structure from ibcore. `selwakeup()` is now done by the `wake_up()` family of functions. https://svnweb.freebsd.org/changeset/base/323350

Add some miscellaneous definitions to support the DRM drivers.
https://svnweb.freebsd.org/changeset/base/322795

Fix for deadlock situation in the LinuxKPI's RCU synchronize API.
https://svnweb.freebsd.org/changeset/base/322746

Use integer type to pass around jiffies and/or ticks values in the LinuxKPI because in FreeBSD ticks are 32-bit.
https://svnweb.freebsd.org/changeset/base/322357

Implement parts of the `hrtimer` API in the LinuxKPI. https://svnweb.freebsd.org/changeset/base/320364

Add `u64_to_user_ptr()` to the LinuxKPI. https://svnweb.freebsd.org/changeset/base/320337

Add `ns_to_ktime()` to the LinuxKPI. https://svnweb.freebsd.org/changeset/base/320336

Add `noop_lseek()` to the LinuxKPI. https://svnweb.freebsd.org/changeset/base/320333

Allow the VM fault handler to be `NULL` in the LinuxKPI when handling a memory map request. When the VM fault handler is `NULL`, a return code of `VM_PAGER_BAD` is returned from the character device's pager populate handler. https://svnweb.freebsd.org/changeset/base/320189

svn UPDATE continued

- Add `kthread` parking support to the LinuxKPI. https://svnweb.freebsd.org/changeset/base/320078

- Add generic `kqueue()` and `kevent()` support to the LinuxKPI character devices. The implementation allows read and write filters to be created and piggybacks on the `poll()` file operation to determine when a filter should trigger. The piggyback mechanism is simply to check for the `EWOULDBLOCK` or `EAGAIN` return code from `read()`, `write()`, or `ioctl()` system calls and then update the `kqueue()` polling state bits. https://svnweb.freebsd.org/changeset/base/319409

- Improve `kqueue()` support in the LinuxKPI. Some applications using `kqueue()` do not set non-blocking I/O mode for event-driven read of file descriptors. This results in the LinuxKPI internal `kqueue` read and write event flags having to be updated before the next read `and/or` write system call, otherwise the read `and/or` write system call may block. https://svnweb.freebsd.org/changeset/base/319501

- Implement 64-bit atomic operations for the LinuxKPI. https://svnweb.freebsd.org/changeset/base/294521

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.

SEE TEXT ONLY

conference REPORT

by Michael W Lucas

BSDCam 2017 Trip Report

Over the decades, FreeBSD development and coordination has shifted from being purely online to involving more and more in-person coordination and cooperation. The FreeBSD Foundation sponsors a devsummit right before BSDCan, EuroBSDCon, and AsiaBSDCon, so that developers traveling to the con can leverage their airfare and hammer out some problems. Yes, the Internet is great for coordination, but nothing beats a group of developers spending 10 minutes together to sketch on a whiteboard and figuring out exactly how to make something bulletproof.

In addition to the coordination efforts, though, conference devsummits are hierarchical. There's a rigid schedule, with topics decided in advance. Someone leads the session. Sessions can be highly informative, passionate arguments, or anything in between.

BSDCam is… a little different. It's an invaluable part of the FreeBSD ecosystem. However, it's something that I wouldn't normally attend.

But right now is not normal.

I'm writing a new edition of *Absolute FreeBSD*. To my astonishment, people have come to rely on this book when planning their deployments and operations. While I find this satisfying, it also increases the pressure on me to get things correct. When I wrote my first FreeBSD book back in 2000, a dozen mailing lists provided authoritative information on FreeBSD development. One person could read every one of those lists. Today, that's not possible—and the mailing lists are only one narrow aspect of the FreeBSD social system.

Don't get me wrong—it's pretty easy to find out what people are doing and how the system works. But it's not that easy to find out what people will be doing and how the system will work. If this book is going to be future-proof, I needed to leave my cozy nest and venture into the wilds of Cambridge, England. Sadly, the BSDCam chair agreed with my logic, so I boarded an aluminum deathtrap—sorry, a "commercial airliner"—and found myself hurtled from Detroit to Heathrow.

And one Wednesday morning, I made it to the William Gates Building of Cambridge University, consciousness nailed to my body by a thankfully infinite stream of proper British tea.

BSDCam attendance is invitation only, and the facilities can only handle 50 folks or so. You need to be actively working on FreeBSD to wrangle an invite. Developers attend from all over the world. Yet, there's no agenda. Robert Watson is the chair, but he doesn't decide on the conference topics. He goes around the room and asks everyone to introduce themselves, say what they're working on, and declare what they want to discuss during the conference. The topics of interest are tallied. The most popular topics get assigned time slots and one of the two big rooms. Folks interested in less popular topics are invited to claim one of the small breakout rooms.

Then the real fun begins.

I started by eavesdropping in the virtualization workshop. For two hours, people discussed FreeBSD's virtualization needs, strengths, and weaknesses. What needs help? What should this interface look like? What compatibility is important, and what isn't? By the end of the session, the couple dozen people had developed a reasonable consensus, and, most importantly, some folks had added items to their to-do lists.

Repeat for a dozen more topics. I got a good grip on what's really happening with security mitigation techniques, FreeBSD's cloud support, TCP/IP improvements, advances in teaching FreeBSD, and more. A BSDCan devsummit presentation on packaging the base system is informative, but eavesdropping on two dozen highly educated engineers arguing about how to nail down the final tidbits needed to make that a real thing is far more educational.

To my surprise, I was able to provide useful feedback for some sessions. I speak at a lot of events outside of the FreeBSD world, and was able to share much of what I hear at Linux conferences. A tool that works well for an experienced developer doesn't necessarily work well for everyone.

Every year, I leave BSDCan tired.

I left BSDCam entirely exhausted. These intense, focused discussions stretched my brain.

But, I have a really good idea where key parts of FreeBSD development are actually headed. This should help future-proof the new *Absolute FreeBSD*, as much as any computer book can be future-proof.

Plus, BSDCam throws the most glorious conference dinner I've ever seen.

I want to thank Robert Watson for his kind invitation, and the FreeBSD Foundation for helping defray the cost of this trip. ●

MICHAEL W LUCAS is the author of several books on FreeBSD, including *Absolute FreeBSD* and the *FreeBSD Mastery* series. Learn more at www.michaelwlucas.com.

FreeBSD™ Journal is published by The FreeBSD Foundation

new faces of FreeBSD

BY DRU LAVIGNE

This column aims to shine a spotlight on contributors who recently received their commit bit and introduces them to the FreeBSD community. This month, the spotlight is on ***Matt Joras***, who received his src commit in July.

Tell us a bit about yourself, your background, and your interests.

I graduated from the University of Illinois at Urbana-Champaign with a BS double major in computer science and physics. I've been working at Isilon since then on the networking team, mostly dealing with the FreeBSD networking stack. I'm interested in a lot of areas of the kernel, and I'm particularly interested (as a long-term goal) in improving/introducing new synchronization primitives into FreeBSD. I also run FreeBSD on my laptop and workstation, so I have a vested interest in improving that overall experience. Areas I think that are of particular importance are the WiFi stack, the Linux compatibility for DRM drivers, and laptop ACPI.

Outside of software, I enjoy hiking, listening to podcasts, following politics, reading fantasy/sci-fi, reading comics, hoarding data, birding, and partaking in the various hobbies my partner enjoys (most recently dog agility competitions).

How did you first learn about FreeBSD and what about FreeBSD interested you?

I've known about the BSDs since I was a teenager running Linux and being generally immersed in the broader open-source community. I had always been skeptical of the GNU-style licenses and thought the BSDs afforded more practical freedoms.

How did you end up becoming a committer?

Contributing to open-source projects is something I wanted to do since I started primarily using open-source software as a teenager. Back then I lacked the tenacity, confidence, and expertise to contribute anything meaningful (or so I thought). Since I abandoned my dreams of being a physicist and decided to pursue software professionally, my ideal has always been to contribute to open source full-time. My job at Isilon isn't exactly that, but it formally introduced me to FreeBSD and allowed me to do work on it. I admired many of the people I work with that were already members of the community, and was enthusiastic to become a part of it.

How has your experience been since joining the FreeBSD Project? Do you have any advice for readers who may be interested in also becoming a FreeBSD committer?

People have been very welcoming! Overall, people are fairly responsive to communications, though it can be difficult to get certain things reviewed or get consensus on how best to fix something. I was extremely lucky in that I was gainfully employed somewhere that uses FreeBSD and has an interest in making the operating system better. If that weren't the case, it would have been much harder to feel as though I knew enough to contribute.

That being said, I think the best way to get involved is to use FreeBSD for as much of your computing as you can. This could be on your servers, your laptop, wherever. The experience is entirely usable, but you will inevitably find bugs and things that could be improved. You can study the relevant areas in the normal fashion, find the appropriate person in the community to get feedback (usually via mailing lists or PRs), and then develop your improvement. If you do this enough, someone will probably take notice and suggest you become a committer. ●

DRU LAVIGNE is a doc committer for the FreeBSD Project and Chair of the BSD Certification Group.

OCT-DEC 2017

BY DRU LAVIGNE

Events Calendar

The following BSD-related conferences will take place during the last quarter of 2017.

OpenZFS Developer Summit • Oct. 24 & 25 • San Francisco, CA

http://open-zfs.org/wiki/OpenZFS_Developer_Summit • The 5th annual OpenZFS Developer Summit will take place at the Children's Creativity Museum in San Francisco. The goal of the event is to foster cross-community discussions of OpenZFS and to make progress on some of the proposed projects. This event consists of a day of presentations followed by a one-day hackathon. There is a nominal registration fee to attend.

LISA • Oct. 29–Nov. 3 • San Francisco, CA

https://www.usenix.org/conference/lisa17 • The 31st Large Installation System Administration Conference will be held in San Francisco. There will be a FreeBSD booth in the expo area, and an expo-only pass is available for a nominal fee.

BSDTW • Nov. 11 & 12 • Taipei, Taiwan

https://bsdtw.org/ • The first annual BSDTW conference aims to encourage more BSD developers throughout Asia, drawing speakers both locally and from the entire world. It will gather in Taipei, Taiwan, for a mix of developer and user-focused presentations, food, and activities.

ARMv8
ENABLED AS TIER 1 ARCHITECTURE:
A COLLABORATIVE DEVELOPMENT PROJECT
Globally Accessible FreeBSD
FFS Filesystem
A Brief History of the BSD Fast Filesystem
Teaching Operating Systems
CHERI
by Marshall Kirk McKusick